Bachelor's Thesis Computing Science

Syntax and Type Checking of Truncated Type Theory

Bram van Veenschoten
s1030516

March 22, 2023

First supervisor/assessor:
Prof. Dr. Herman Geuvers

Second assessor:
Dr. Niels van der Weide

Radboud University

Abstract

Dependent type systems are formal languages that can serve as foundations for programming languages and theorem proving systems. The Calculus of Constructions (CC) in particular is a dependent type system which has a lot of expressive power, relative to its simplicity. CC admits encodings of datatypes such as booleans, natural numbers and lists, but leaves their corresponding induction principles to be desired. Certain useful extensionality principles are also not derivable. This thesis introduces Truncated Type Theory (TTT) as an extension of CC, using concepts from Cubical Type Theory. We show that Function Extensionality is derivable. Furthermore, we present a bidirectional type checking algorithm for TTT.

Contents

Chapter 1

Introduction

Dependent type systems are formal languages for programming and logic. Proof assistants based on dependent type theory allow their users to write programs and proofs about them, and the system checks whether the proofs are correct. The validity of the proofs then depends on the reliability of the proof checker. The smaller and simpler the checker is, the more confidence one may have that it is correct. This software engineering principle is called the De Bruijn Criterion [6]. Of course, checking a proof is only simple if the rules of the type system are. This means that the implementation of a proof-assistant requires a trade-off between simplicity and expressivity of the underlying type system.

Induction is one concept that is arguably essential, as it allows the definitions of data types such as the natural numbers, lists and trees. On top of inductive types one can build additional features such as indexed inductive types, and functions defined by dependent pattern matching and structural recursion. Modern proof assistants based on dependent type theory such as Coq [24], Agda [26] and Lean [15] implement these features to various extents. They have indexed inductive type schemas as part of the trusted kernel. Users can specify data types from a number of constructors. The system then checks whether the specification is well-founded and the indices are well-formed. After that it derives appropriate induction principles. The necessary checks are quite involved, and add significant complexity to a system that would otherwise be simple.

The complexity of conventional datatype systems motivates our search for an alternative. The Calculus of Constructions (CC)[14] is a dependent type system in which one can encode inductive types, but their corresponding induction principles are not derivable. Awodey et al. [4] have shown that the induction principles are provable if the CC is extended with a few concepts from Homotopy Type Theory (HoTT). The particular features they use are the axiom of Function Extensionality (FunExt) and small dependent sum types.

However, the theory used by Awodey et al. is not satisfactory because it relies on axioms that do not compute. In dependent type theory, terms are computationally equal if they compute to the same term. Computational equality requires no proof, which means reasoning with computational equality is effectively automated. On the other hand, axioms can block computation, so equational reasoning involving axioms does require proof, which can be quite cumbersome. We would like to augment the theory so that FunExt does not get in the way of reduction. This is the main problem we address in this thesis.

We use concepts from Cubical Type Theory [12], which has been developed to implement FunExt and the univalence axiom with computational content. The latter axiom identifies types that are isomorphic. Implementing computational univalence comes at the cost of adding complexity to the type system. We make different trade-off, favouring simplicity. Fortunately, the constructions of Awodey et al. do not require univalence. We observe that if we leave univalence out of Cubical Type Theory, we are left with a simple type system wherein induction is derivable, and computation does not get blocked by axioms. This system we call Truncated Type Theory (TTT).

Overview We give an introduction to the Calculus of Constructions in Chapter 2. In Chapter 3 we discuss approaches and problems related to equality in dependent type theory. We present and explain Truncated Type Theory (TTT) in Chapter 4. We discuss some meta-theoretical properties of TTT in Chapter 5. In Chapter 6 we present a bidirectional type checking algorithm for TTT, with some novel features specific to TTT.

Chapter 2

Preliminaries

We give a brief overview of Type Theory. For a more detailed treatment we refer to [6, 8, 17]. The concepts are quite abstract and may be hard to grasp when presented directly. We attempt to give a gentle introduction by presenting it as a series of extensions to a simpler system. We first present the Simply Typed Lambda Calculus (STLC), then we gradually introduce extensions to System F, System Fω, and finally the Calculus of Constructions.

2.1 The Simply Typed Lambda Calculus

The Simply Typed Lambda Calculus (STLC) was originally introduced by Church [11]. The terms and types of the STLC are defined as follows.

$$\begin{array}{rll} A, B, C \in \text{Type} ::= & X & \text{type variable} \\ & \mid A \to B & \text{function type} \end{array}$$

$$\begin{array}{rll} f, g, t, u \in \text{Term} ::= & x & \text{variable} \\ & \mid \lambda x.t & \text{abstraction} \\ & \mid f\ t & \text{application} \end{array}$$

Before we define the rules for determining the correct type of a term, we need a notion of contexts, assigning types to variables.

$$\begin{array}{rll} \Gamma \in \text{Context} ::= & \cdot & \text{empty context} \\ & \mid \Gamma, x : A & \text{context extension} \end{array}$$

Now, we define the typing judgment.

$$\Gamma \vdash t : A \qquad \text{In context } \Gamma, \text{ the term } t \text{ has type } A$$

The typing rules are defined as follows.

$$\frac{\Gamma, x : A \vdash t : B}{\Gamma \vdash \lambda x.t : A \to B}\ \textsc{Lam} \qquad \frac{x : A \in \Gamma}{\Gamma \vdash x : A}\ \textsc{Var}$$

$$\frac{\Gamma \vdash t : A \to B \qquad \Gamma \vdash u : A}{\Gamma \vdash t\ u : B}\ \textsc{App}$$

The Var rule states that a variable is well typed if it is in the context. The App rule states that an application $t\ u$ is well typed if t has a function type, and the type of u is the domain of t. The whole application then has as type the range of t. Finally, the Lam rule states that a λ-abstraction has a function type if, in the context extended by the bound variable x with the function domain A, the body t has type B.

The STLC has a notion of computation, which allows it to serve as a foundation for programming languages. Computation in λ-calculi is defined by β-reduction. A single step of computation is defined by the following rule.

$$\frac{}{(\lambda x.t)u \Rightarrow t[x := u]}\ \textsc{Beta}$$

The Beta rule states that an abstraction $\lambda x.t$ applied to an argument u reduces to its body $t[x := u]$. We write $t[x := u]$ to mean the usual capture avoiding substitution. We continue the use of the notation $t \Rightarrow t'$ to denote computation rules.

Readers familiar with natural deduction may notice a similarity between the Lam, App and Var rules and Implication Introduction, Implication Elimination and Assumption rules, respectively. Indeed, STLC can be used as a deduction system, where we interpret types as propositions, in particular function types as implications, and terms as proofs. STLC may be extended with constructs corresponding to truth, falsehood, disjunction and conjuction, but these are subsumed by the extensions described in the next section, so we do not discuss them here. The notion that types correspond to propositions is called the Curry-Howard correspondence, after the persons who discovered it [21].

As an example, we construct the derivation for function composition, which corresponds to syllogism in logic.

Example 1. Let A,B and C be arbitrary types/propositions. We construct function composition/a proof of syllogism.

$$\cdot \vdash \lambda x.\lambda y.\lambda z.x\ (y\ z) : (B \to C) \to (A \to B) \to A \to C$$

Our example works independently of the types we choose for A, B and C. If we want to state this fact in our deduction system, we need to extend it with a notion of universal quantification, as we discuss in the next section.

2.2 System F

If we extend STLC with universal quantification we get the Polymorphic λ-calculus, or System F, introduced independently by Girard [19, 18] and Reynolds [29]. We extend the syntax of types with variables and quantification.

$$\begin{array}{rll} A, B, C \in \text{Type} ::= & X & \text{type variable} \\ & \mid A \to B & \text{function type} \\ & \mid \forall X.A & \text{quantification} \end{array}$$

We extend terms with constructors and eliminators for quantification.

$$\begin{array}{rll} f, t, u \in \text{Term} ::= & x & \text{variable} \\ & \mid \lambda x.t & \text{abstraction} \\ & \mid f\ t & \text{application} \\ & \mid \Lambda X.t & \text{generalization} \\ & \mid t[A] & \text{instantiation} \end{array}$$

Finally, we extend contexts with type variables.

$$\begin{array}{rll} \Gamma \in \text{Context} ::= & \cdot & \text{empty context} \\ & \mid \Gamma, x : A & \text{context extension} \\ & \mid \Gamma, X & \text{type variable extension} \end{array}$$

We remark that our presentation is non-standard, but in the style of Domain-free Pure Type Systems [9]. This presentation corresponds more intuitively with our type checking algorithm in Chapter 6. We extend the typing rules of the STLC with rules for instantiation and generalization, defined as follows.

$$\frac{\Gamma, X \vdash t : A \qquad X \notin \Gamma}{\Gamma \vdash \Lambda X.t : \forall X.A}\ \textsc{Gen} \qquad\qquad \frac{\Gamma \vdash t : \forall X.A}{\Gamma \vdash t[B] : A[X := B]}\ \textsc{Inst}$$

Now we can restate our composition example.

Example 2. Polymorphic composition/syllogism

$$\cdot \vdash \Lambda X.\Lambda Y.\Lambda Z.\lambda x.\lambda y.\lambda z.x\ (y\ z) : \forall X.\forall Y.\forall Z.(Y \to Z) \to (X \to Y) \to X \to Z$$

The fact that universal quantification is itself a type makes System F very powerful. For example, we can encode the connectives of propositional logic missing from the previous section. Functional programmers may recognize conjunction and disjunction correspond to 2-tuples and the Either type, respectively. We also encode natural numbers in System F.

Example 3. Conjunction $A \wedge B$ is encoded as $\forall X.(A \to B \to X) \to X$. We define the introduction- and elimination rules.

$$\frac{\Gamma \vdash t : A \qquad \Gamma \vdash u : B}{\Gamma \vdash \texttt{con-intro}(t, u) \equiv \Lambda X.\lambda x.x\ t\ u : A \wedge B}$$

$$\frac{\Gamma \vdash t : A \wedge B}{\Gamma \vdash \texttt{fst}(t) \equiv t[A](\lambda x.\lambda y.x) : A} \qquad \frac{\Gamma \vdash t : A \wedge B}{\Gamma \vdash \texttt{snd}(t) \equiv t[B](\lambda x.\lambda y.y) : B}$$

Example 4. Disjunction $A \vee B$ is encoded as $\forall X.(A \to X) \to (B \to X) \to X$. We define the introduction- and elimination rules.

$$\frac{\Gamma \vdash t : A}{\Gamma \vdash \texttt{inl}(t) \equiv \Lambda X.\lambda x.\lambda y.x\ t : A \vee B} \qquad \frac{\Gamma \vdash t : B}{\Gamma \vdash \texttt{inr}(t) \equiv \Lambda X.\lambda x.\lambda y.y\ t : A \vee B}$$

$$\frac{\Gamma \vdash t : A \vee B \qquad \Gamma \vdash f : A \to C \qquad \Gamma \vdash g : B \to C}{\Gamma \vdash \texttt{dis-elim}(f, g, t) \equiv t[C] f\ g : C}$$

Example 5. Negation $\neg A$ is encoded as $A \to (\forall X.X)$. We define the elimination rule.

$$\frac{\Gamma \vdash t : \neg A \qquad \Gamma \vdash q : A}{\Gamma \vdash \texttt{neg-elim}(t, q) \equiv t\ q[B] : B}$$

Example 6. We encode $\mathbb{N}$ as $\forall X.X \to (X \to X) \to X$. Given a type C, a term $t : C$ and a function $f : C \to C$ and the encoding of a natural number n, the term $n[C]\ t\ f$ applies f to t, n times. We define some numerals and addition.

$$\begin{aligned} 0 &\equiv \Lambda X.\lambda x.\lambda y.x \\ 1 &\equiv \Lambda X.\lambda x.\lambda y.y\ x \\ 2 &\equiv \Lambda X.\lambda x.\lambda y.y\ (y\ x) \\ 3 &\equiv \Lambda X.\lambda x.\lambda y.y\ (y\ (y\ x)) \\ n + m &\equiv \Lambda X.\lambda x.\lambda y.n[X]\ (m[X]\ x\ y)\ y \end{aligned}$$

2.3 System Fω

In the previous section, we showed how to encode the conjunction or disjunction of two propositions. In our example we informally made use of type operators. Conjunction and disjunction can be seen as operators (or functions) that take two types as arguments, and produce a new one. This kind of reasoning can be quite useful, when incorporated in System F, we get a type system called Fω, also described in [19]. The type operators in Fω are subject to a kinding-discipline similar to the typing rules of STLC, to ensure they are well-formed. We refer to the expressions denoting proper types and type functions as 'constructors'. We define the syntax of kinds.

$$\begin{aligned} K \in \text{Kind} ::= {} & * && \text{The kind of proper types} \\ \mid {} & K_1 \to K_2 && \text{type function} \end{aligned}$$

We extend the syntax of types with functions and applications. Quantifications now specify the kind of the type they quantify over.

$$
\begin{array}{rll}
A, B, C \in \text{Constructor} ::= & X & \text{constructor variable} \\
& \mid \lambda x.A & \text{constructor abstraction} \\
& \mid A\ B & \text{constructor application} \\
& \mid A \to B & \text{function type} \\
& \mid \forall X : K.A & \text{quantification}
\end{array}
$$

We change the syntax of contexts so that constructors specify their kind.

$$
\begin{array}{rll}
\Gamma \in \text{Context} ::= & \cdot & \text{empty context} \\
& \mid \Gamma, x : A & \text{context extension} \\
& \mid \Gamma, X : K & \text{constructor variable extension}
\end{array}
$$

Now, we need judgments to ensure constructors and contexts are well-formed.

$$
\begin{array}{rl}
\Gamma \vdash \mathbf{ctx} & \text{The context } \Gamma \text{ is well-formed} \\
\Gamma \vdash A : K & \text{The constructor } A \text{ has kind } K
\end{array}
$$

The well-formedness rules for contexts and constructors are defined as follows. Note that the rules of the STLC are still valid, while those of System F are replaced.

$$\frac{}{\cdot \vdash \mathbf{ctx}}\ \textsc{EmptyCtx}$$

$$\frac{\Gamma \vdash \mathbf{ctx} \qquad \Gamma \vdash A : * \qquad x \notin \Gamma}{\Gamma, x : A \vdash \mathbf{ctx}}\ \textsc{ExtendTerm}$$

$$\frac{\Gamma \vdash \mathbf{ctx} \qquad X \notin \Gamma}{\Gamma, X : K \vdash \mathbf{ctx}}\ \textsc{ExtendCon}$$

$$\frac{\Gamma \vdash \mathbf{ctx} \qquad X : K \in \Gamma}{\Gamma \vdash X : K}\ \textsc{ConVar}$$

$$\frac{\Gamma \vdash A : K_1 \to K_2 \qquad \Gamma \vdash B : K_1}{\Gamma \vdash A\ B : K_2}\ \textsc{ConApp}$$

$$\frac{\Gamma, X : K_1 \vdash A : K_2}{\Gamma \vdash \lambda X.A : K_1 \to K_2}\ \textsc{ConLam}$$

$$\frac{\Gamma \vdash A : * \qquad \Gamma \vdash B : *}{\Gamma \vdash A \to B : *}\ \textsc{Fun}$$

$$\frac{\Gamma, X : K \vdash A : *}{\Gamma \vdash \forall X : K.A : *}\ \textsc{Forall}$$

The fact that types can be computed by simple programs means there are multiple ways to express one type. For example, we want the types $(\lambda X.X)A$ and A to be considered one and the same, because applying β-reduction to the former yields the latter. Therefore, Fω has a notion of convertible types, which we consider to be interchangeable.

$$\frac{\Gamma \vdash t : A \qquad \Gamma \vdash B : * \qquad \Gamma \vdash A \equiv B : *}{\Gamma \vdash t : B}\ \textsc{Conv}$$

We remark that our presentation of conversion is not standard, because we include kinding information in the judgments. This is consistent with our presentation in Chapter 4, where it is necessary. The conversion relation is reflexive, symmetric and transitive. Of course, it also includes β-reduction.

$$\frac{\Gamma \vdash A : K}{\Gamma \vdash A \equiv A : K}\ \textsc{Refl} \qquad\qquad \frac{\Gamma \vdash A \equiv A' : K}{\Gamma \vdash A' \equiv A : K}\ \textsc{Sym}$$

$$\frac{\Gamma \vdash A \equiv A' : K \qquad \Gamma \vdash A' \equiv A'' : K}{\Gamma \vdash A \equiv A'' : K}\ \textsc{Trans}$$

$$\frac{\Gamma \vdash A, B : K \qquad A \Rightarrow B}{\Gamma \vdash A \equiv B : K}\ \textsc{RedConv}$$

Finally, we have congruence rules stating that compound expressions are convertible if their components are.

$$\frac{\Gamma, X : K \vdash A \equiv A' : *}{\Gamma \vdash \forall X : K.A \equiv \forall X : K.A' : *}\ \textsc{ForallCong}$$

$$\frac{\Gamma \vdash A \equiv A' : * \qquad \Gamma \vdash B \equiv B' : *}{\Gamma \vdash A \to B \equiv A' \to B' : *}\ \textsc{FunCong}$$

$$\frac{\Gamma \vdash A \equiv A' : K_1 \to K_2 \qquad \Gamma \vdash B \equiv B' : K_1}{\Gamma \vdash A\ B \equiv A'\ B' : K_2}\ \textsc{TyAppCong}$$

$$\frac{\Gamma, X : K_1 \vdash A \equiv A' : K_2}{\Gamma \vdash \lambda X.A \equiv \lambda X.A' : K_1 \to K_2}\ \textsc{TyLamCong}$$

Now we may write abstractions over higher-kinded structures. For example, we encode the Haskell language's notion of functor in Fω

Example 7. We encode functoriality as follows. We show the identity function on types is a Functor.

$$\begin{aligned} \texttt{Functor} &:= \lambda F.\forall X.\forall Y.(X \to Y) \to F\ X \to F\ Y && : (* \to *) \to * \\ \texttt{map-id} &:= \Lambda X.\lambda Y.\lambda x.x && : \texttt{Functor}\ (\lambda X.X) \end{aligned}$$

2.4 The Calculus of Constructions

So far, System Fω allows us to do propositional reasoning, write functional programs that compute types, and universally quantify over types. If we want to reason about the regular programs we write, we need to extend the Curry-Howard isomorphism to predicate logic. If $*$ is the type of propositions, then for a type A, a predicate P on A should have kind $A \to *$. Furthermore, we should have universal quantification $\forall x : A.P\ x : *$. Note how kinds must now include functions out of types, and how similar quantification over terms is to quantification over types. Also recall the similarities between the kinding rules of Fω and the typing rules of STLC. Instead of extending the syntax yet again, we collapse the languages of terms, types and kinds into one language of terms. The resulting system is the Calculus of Constructions (CC), introduced by Coquand and Huet [14], also referred to as λC. The dependent product type $\prod$ now acts as universal quantification over types and kinds, as well as the type of functions. Simple functions are now a special case of dependent functions, and we continue to write $A \to B$ for when the bound variable does not occur in B. The typing of terms and the kinding of types is now determined by the same rules. We add a sort $\Box$ to act as the 'type' of kinds. We define the syntax below.

$$
\begin{array}{rll}
f, t, u, A, B \in \text{Term} ::= & * & \text{kind of types} \\
| & \Box & \text{superkind of kinds} \\
| & x & \text{variable} \\
| & \prod x : A.B & \text{dependent product type} \\
| & \lambda x.t & \text{abstraction} \\
| & t\ u & \text{application}
\end{array}
$$

The typing- and kinding rules for quantification and function types are all subsumed by the dependent product rule. We define a set of sorts $\mathcal{S}$ and a set of kinding rules $\mathcal{R}$ determining functions between which sorts are allowed. This is in line with the pure type system presentation of Barendregt [5]. For now the rules allow domains and ranges of any sort, but that will change as we extend CC.

$$\mathcal{S} = \{*, \Box\}$$

$$\mathcal{R} = \{(*, *, *), (*, \Box, \Box), (\Box, *, *), (\Box, \Box, \Box)\}$$

$$\frac{\Gamma \vdash A : s_0 \qquad \Gamma, x : A \vdash B : s_1 \qquad (s_0, s_1, s_2) \in \mathcal{R}}{\Gamma \vdash \prod x : A.B : s_2}\ \textsc{Pi}$$

$$\frac{\Gamma \vdash t : \prod x : A.B \qquad \Gamma \vdash u : A}{\Gamma \vdash t\ u : B[x := u]}\ \textsc{App}$$

The remainder of the typing- and conversion rules are very similar to those of Fω, so we omit them here. They can be found in Appendix B.1 and D (in addition to some extensions). Fω has a syntactical distinction between terms, constructors and kinds. Although in CC these are collapsed into a single syntactic category, the stratification between them is preserved by the typing system.

Definition 1. We define the sets of objects, constructors and kinds of CC

$$\begin{aligned}\text{Object} &:= \{t \in \text{Term} \mid \exists \Gamma, A[\Gamma \vdash A : * \wedge \Gamma \vdash t : A]\}\\ \text{Constructor} &:= \{A \in \text{Term} \mid \exists \Gamma, K[\Gamma \vdash K : \Box \wedge \Gamma \vdash A : K]\}\\ \text{Kind} &:= \{K \in \text{Term} \mid \exists \Gamma[\Gamma \vdash K : \Box]\}\end{aligned}$$

Note that these sets are not disjoint.

A new thing we can do in CC is equational reasoning with Leibniz equality.

Example 8. Leibniz' principle of the identity of indiscernables states that two objects are equal if they have the same properties. We can encode this in CC.

$$x =_A y := \prod P : A \to *.P\ x \to P\ y : *$$

Suppose $A : *$ and $x, y, z : A$. We show equality is reflexive, symmetric, transitive and substitutive. Symmetry may be surprising given the definition, but it holds because our definition of equality is included in the type of predicates it quantifies over.

$$\begin{aligned}\texttt{refl} &:= \lambda P.\lambda p.p && : x =_A x\\ \texttt{sym} &:= \lambda y.\lambda e.e\ (\lambda z.z =_A x)\ \texttt{refl} && : x =_A y \to y =_A x\\ \texttt{trans} &:= \lambda e.\lambda e'.\lambda P.\lambda p.e'\ P\ (e\ P\ p) && : x =_A y \to y =_A z x =_A z\\ \texttt{subst} &:= \lambda e.e && : x =_A y \to \prod P : A \to *.P\ x \to P\ y\end{aligned}$$

Functors are supposed to obey certain laws. With Leibniz equality, we can encode these in CC. Our identity functor satisfies these laws.

Example 9. Let $F : * \to *$ be a functor and $A : *$. We define the identity law for F as follows.

$$\texttt{F-id} : \texttt{map}\ A\ A\ (\lambda x.x) =_{F\ A \to F\ A} \lambda x.x$$

Let $F : * \to *$ be a functor, $A, B, C : *$, $f : A \to B$ and $g : B \to C$. The composition law is defined as follows.

$$\texttt{F-comp} : \lambda x.\texttt{map}\ B\ C\ g\ (\texttt{map}\ A\ B\ f\ x) =_{F\ A \to F\ C} \texttt{map}\ A\ C\ (\lambda x.g\ (f\ x))$$

While CC affords a lot of expressive power, it leaves some things to be desired. For one, Leibniz equality does not satisfy Function Extensionality (FunExt) or Uniqueness of Identity Proofs (UIP). That is, the following types are not inhabited.

$$\texttt{funext} := (\prod x : A. f\ x =_{B\ x} g\ x) \to f =_{\prod x:A. B\ x} g$$
$$\texttt{uip} := \prod e : x =_A x. e =_{x=_A x} \texttt{refl}$$

Another drawback is that induction principles of impredicative encodings are not derivable.

2.5 Universes & Impredicativity

The sort $*$ in CC is closed under quantification, meaning that for any $\Gamma \vdash A : s$ and $\Gamma, x : A \vdash B : *$ we have $\Gamma \vdash \prod x : A.B : *$ regardless of the sort s. In type theory, this feature is called impredicativity. In contrast, in predicative systems a type $\prod x : A.B$ inhabits a sort no smaller than either of the sorts of A or B. Predicative type systems may have an infinite hierarchy of sorts $\Box_i$ indexed by natural numbers, with the following axiom and dependent product formation rule.

$$\mathcal{A} := \{(\Box_i, \Box_{i+1}\} \qquad\qquad \mathcal{R} := \{(\Box_i, \Box_j, \Box_{\max(i,j)})\}$$

Impredicative systems have more expressive power compared to predicative ones, for example the Church-encoding of datatypes cannot be encoded in predicative type theories. This expressivity does come at the cost of being much harder to reason about the meta-theory of impredicative systems, as well as them being incompatible with a number of other features and axioms.

Chapter 3

Equality in Dependent Type Theory

As stated previously, equality in CC leaves Function Extensionality to be desired. In this section we explore several approaches to equality that have been developed.

3.1 Martin Löf Type Theory

Martin Löf Type Theory [23, 25] is a predicative type theory, variants of which generally come in two flavours: Extensional (ETT) and Intensional (ITT). ITT has an equality type as a primitive, along with introduction- and elimination rules.

$$\frac{\Gamma \vdash A : U_i \qquad \Gamma \vdash t, u : A}{\Gamma \vdash t =_A u : U_i} \qquad \frac{\Gamma \vdash A : U_i \qquad \Gamma \vdash t : A}{\Gamma \vdash \mathbf{refl}\ t : t =_A t}$$

$$\frac{\begin{array}{c}\Gamma \vdash A : U_i \qquad \Gamma \vdash t, u : A \qquad \Gamma \vdash p : t =_A u \\ \Gamma \vdash P : \prod x : A.t =_A u \to U_j \qquad \Gamma \vdash v : P\ t\ (\mathbf{refl}\ t)\end{array}}{\Gamma \vdash J(A, t, u, p, P, v) : P\ u\ p}$$

$$\frac{\begin{array}{c}\Gamma \vdash A : U_i \qquad \Gamma \vdash t : A \\ \Gamma \vdash P : \prod x : A.t =_A u \to U_j \qquad v : P\ t\ (\mathbf{refl}\ t)\end{array}}{J(A, t, t, \mathbf{refl}\ t, P, v) \Rightarrow v}$$

Although the J-rule in ITT is slightly stronger than Leibniz equality in CC, ITT also does not have UIP or FunExt. ETT has the same rules as ITT described above, with additionally an equality reflection rule, stating that two terms are convertible if there is a proof of their equality.

$$\frac{\Gamma \vdash e : t =_A u}{\Gamma \vdash t \equiv u}$$

UIP and FunExt are derivable in ETT. Equality reflection adds a lot of strength to type theory, arguably too much, because it also breaks a number of desirable meta-theoretical properties, such as normalization and decidability of type-checking [20].

3.2 Observational Type Theory

Observational type Theory (OTT) [2] has been developed with the aim of affirming UIP and FunExt like ETT, while retaining decidability of type-checking like ITT. It features equality between terms and types, as well as casts between provably equal types.

$$\frac{\Gamma \vdash A, B : U}{\Gamma \vdash A =_U B : U} \qquad \frac{\Gamma \vdash A : U \qquad \Gamma \vdash t, u : A}{\Gamma \vdash t =_A u : U}$$

$$\frac{\Gamma \vdash A, B : U \qquad \Gamma \vdash e : A =_U B \qquad \Gamma \vdash t : A}{\Gamma \vdash \mathbf{cast}(A, B, e, t) : B}$$

The equality type in OTT is actually an eliminator that computes on the type structure, relying on the fact that universes U are closed. Similarly, cast also computes on the type structure. For example, equality between two dependent product types reduces to a pair of equalities of the domains and ranges, respectively. A cast between two function types reduces to a cast over the domain and range equalities.

$$\frac{\Gamma \vdash A, A' : U \qquad \Gamma, x : A \vdash B : U \qquad \Gamma, x' : A \vdash B' : U}{\begin{array}{c}(\prod x : A.B) =_U (\prod x' : A'.B') \Rightarrow \\ \sum e : A' =_U A.\prod x' : A'.B\ (\mathbf{cast}(A', A, e, x')) =_U B' x'\end{array}}$$

$$\frac{\begin{array}{lll}\Gamma \vdash A, A' : U & \Gamma, x : A \vdash B : U & \Gamma, x' : A' \vdash B' : U \\ \Gamma \vdash e : (\prod x : A.B) =_U (\prod x' : A'.B') & \Gamma \vdash t : \prod x : A.B & x'' := \mathbf{cast}(A', A, e.\pi_1, x')\end{array}}{\Gamma \vdash \mathbf{cast}(\prod x : A.B, \prod x' : A'.B', e, t) \Rightarrow \lambda x'.\mathbf{cast}(B[x := x''], B'e.\pi_2\ x', t\ x'')}$$

Equality on functions reduces to point-wise equality, hence FunExt holds in OTT.

$$\frac{\Gamma \vdash A : U \qquad \Gamma, x : A \vdash B : U \qquad \Gamma \vdash f, g : \prod x : A.B}{f =_{\prod x:A.B} g \Rightarrow \prod x : A.f =_B g}$$

UIP is simply a conversion rule in OTT. While predicative variants of OTT have been shown to be normalizing, [27], this property does not extend to impredicative sorts. Abel and Coquand presented a counterexample to normalization in impredicative type theory with a proof-irrelevant coercion operator [1]. Pujet and Tabareau have shown normalization in OTT may be preserved with a proof-irrelevant impredicative sort [28], but types in such a sort cannot encode programs, only logical propositions. For the purposes of adding FunExt and UIP to CC, observational equality turns out to be too strong still.

3.3 Cubical Type Theory

In a different line of work, Cubical Type Theories [12] were developed to give a computational interpretation of Homotopy Type Theory (HoTT) [31] and the Univalence Axiom (UA). The latter states that two types are equal if they are equivalent.

$$(A = B) \simeq (A \simeq B)$$

Here, 'equivalence' can mean a one-to-one correspondence, or some equivalent notion. UA can be seen as an extensionality principle for the universe, but it is not consistent with UIP [20].

In order to provide a computational interpretation of UA, Cubical Type Theories extend ITT with a number of features. The first is an abstract interval $\mathbf{I}$ with two constant values $\mathbf{i}_0$ and $\mathbf{i}_1$. Functions out of the interval are interpreted as lines, and lines with fixed endpoints are called paths. Paths induce a notion of heterogeneous equality in cubical type theories.

Second is Kan operations to compose and transport along paths. A detailed discussion of these is out of scope for this thesis. We note that transport along paths computes on the structure of types, similarly to cast in OTT. We will be using a simplified notion of transport for Truncated Type Theory in the next section.

The third feature of cubical type theories is glue types to embed equivalences into paths, thus adding a computational interpretation of the univalence axiom. The details are very complex and out of scope of this thesis. XTT [30] is a variant of cubical type theory without glue types or univalence, instead validating UIP through boundary separation. Similarly, we have no need for univalence for our purposes, and more use for UIP. Our theory is very similar to XTT in that regard.

Chapter 4

Truncated Type Theory

We define Truncated Type Theory by extending CC with constructs inspired by cubical Agda [32]. This section introduces and explains the new constructs. The full system of TTT is in the Appendix.

4.1 The Interval & Equality type

Equality in TTT depends on an abstract interval $\mathbf{I}$, which has constant endpoints $\mathbf{i}_0$ and $\mathbf{i}_1$:

$$\frac{}{\Gamma \vdash \mathbf{I} : \square_{\mathbf{I}}}\ \mathrm{I} \qquad \frac{}{\Gamma \vdash \mathbf{i}_0 : \mathbf{I}}\ \mathrm{I0} \qquad \frac{}{\Gamma \vdash \mathbf{i}_1 : \mathbf{I}}\ \mathrm{I1}$$

The interval type comes with its own special sort. Dependent function formation with this sort is restricted, we can make functions out of the interval, but standard CC-terms cannot eliminate into the interval. It should be noted that in cubical type theories, the interval has a geometric interpretation as a line. While TTT borrows the cubical syntax, the semantics of the interval do not carry over to TTT. The following are the type formation, introduction and elimination rules for our equality type:

$$\frac{\Gamma \vdash A : \mathbf{I} \to * \qquad \Gamma \vdash t : A\ \mathbf{i}_0 \qquad \Gamma \vdash u : A\ \mathbf{i}_1}{\Gamma \vdash t =_A u : *}\ \textsc{Eq}$$

$$\frac{\Gamma \vdash A : \mathbf{I} \to * \qquad \Gamma \vdash p : \prod i : \mathbf{I}.A\ i}{\Gamma \vdash \lfloor p \rfloor : (p\ \mathbf{i}_0) =_A (p\ \mathbf{i}_1)}\ \textsc{EqIntro}$$

$$\frac{\begin{array}{ll}\Gamma \vdash A : \mathbf{I} \to * & \Gamma \vdash t : A\ \mathbf{i}_0 \\ \Gamma \vdash u : A\ \mathbf{i}_1 & \Gamma \vdash e : t =_A u \qquad \Gamma \vdash i : \mathbf{I}\end{array}}{\Gamma \vdash \mathbf{Elim}_{=}(A, t, u, e, i) : A\ i}\ \textsc{EqElim}$$

The eliminator has computation rules:

$$\frac{}{\mathbf{Elim}_{=}(A, x, y, e, \mathbf{i}_0) \Rightarrow x}\ \textsc{EqElim0}$$

$$\frac{}{\mathbf{Elim}_{=}(A, x, y, e, \mathbf{i}_1) \Rightarrow y}\ \textsc{EqElim1}$$

The introduction rule states that for any function out of the interval p, its applications to the endpoints $\mathbf{i}_0$ and $\mathbf{i}_1$ are equal: $p\ \mathbf{i}_0 =_A p\ \mathbf{i}_1$. Conversely, the eliminator allows us to construct a function out of the interval from an equality proof. This function, when applied to an endpoint will return either equated term. As such the interval allows us to 'view' an equation from a particular side.

This idea is also used in the equality type itself. For a function $A : \mathbf{I} \to *$, we can equate two terms of the types $A\ \mathbf{i}_0$ and $A\ \mathbf{i}_1$. In this way equalities can depend on each other, so we can equate terms of types that are provably equal even if they are not convertible. Similarly to function types, a non-dependent equality can be encoded by a constant type function: When we have $t, u : A$ we can form the equality type $t =_{\lambda\_.A} u$. We introduce abbreviations for the equality type and its eliminator.

Notation 1. We write $t =_{\lambda\_.A} u$ as $t = u$ if A can be inferred.

Notation 2. We write $\mathbf{Elim}_{=}(A, t, u, e, i)$ as $e@i$ if the type of e can be inferred.

4.2 Proving with cubical equality

With the rules introduced so far, we can already prove some interesting theorems. For an arbitrary term t, we prove $t = t$ by applying the equality introduction rule on a constant function returning t.

Example 1. Let $t : A$. We have:

$$\lfloor \lambda\_.t \rfloor : t = t$$

We also prove congruence of application.

Example 2. Let $f, f' : \prod x : A.B$ be functions and let $e : f = f'$ be a proof of their equality. Let $t, t' : A$ be terms of the accepted by f, f' and $e' : t = t'$ a proof of their equality, we prove that their respective applications are equal.

$$\lfloor \lambda i.(e@i)(e'@i) \rfloor : f\ t =_{\lambda i.B\ (e@i)} f'\ t'$$

Note that $f\ t$ and $f'\ t'$ have different types, so we use a heterogeneous equality here.

One important property of our cubical presentation is function extensionality is now derivable as a theorem.

Example 3. Let $f, f' : \prod x : A.B$ be functions and $e : \prod x : A.f\ x = f'x$ a proof of their point wise equality. We prove $f = f'$.

$$\lfloor \lambda i.\lambda x.e\ x\ i \rfloor : f = f'$$

The rules presented so far are not sufficient to derive the Leibniz substitution property like we have in CC or MLTT. For that, we introduce a coercion operation that casts terms between provably equal types. When casting between convertible types, coercion is the identity:

$$\frac{\Gamma, i : \mathbf{I} \vdash A : s \qquad \Gamma \vdash t : A[i := \mathbf{i}_0] \qquad s \in \{*, \square\}}{\Gamma \vdash [i.A] \triangleright t : A[i := \mathbf{i}_1]}\ \textsc{Coe}$$

$$\frac{\begin{array}{cc}\Gamma, i : \mathbf{I} \vdash A : s & s \in \{*, \square\} \\ \Gamma \vdash t : A[i := \mathbf{i}_0] & \Gamma \vdash A[i := \mathbf{i}_0] \equiv A[i := \mathbf{i}_1] : s\end{array}}{\Gamma \vdash [i.A] \triangleright t \equiv t : A[i := \mathbf{i}_1]}\ \textsc{CoeId}$$

Now we can derive the Leibniz substitution principle:

Example 4. Let $t, u : A$ be terms and $e : t = u$ a proof of their equality. Also, let $P : A \to *$ and $p : P\ t$ be a predicate with a proof that it holds for t. We prove $P\ u$.

$$[(\lambda i.P\ (e@i)).p]\triangleright : P\ u$$

4.3 Type-directed coercion

We would like for coercion to reduce by recursion on the type of the coerced term, similarly to OTT. This means we need reduction rules for coercion for each type-former, those being $\prod$, $\sum$ and $*$. Before we can do that, we need some additional manipulations on points of the interval. We introduce inverse $(-)$, minimum $(\wedge)$ and maximum $(\vee)$ operations, which together form a De Morgan algebra. The De Morgan algebra states the equalities one would expect for the interval with $\mathbf{i}_0$ as the minimum and $\mathbf{i}_1$ as the maximum element. For the typing- and conversion rules, we refer the interested reader to the Appendix. The inverse operator maps the constant interval points to their counterpart, and so allows us to invert the view on an equality. Effectively, we can use it to show symmetry of equality:

Example 5. Let $t, u : A$ be two terms and $e : t = u$ be a proof of their equality. We prove:

$$\lfloor \lambda i.e@(-i) \rfloor : u = t$$

Now we can define type-directed reduction rules for coercion. First of all, coercion of sorts is simply always the identity:

$$\frac{}{[i.\mathbf{I}*] \triangleright t \Rightarrow t}\ \textsc{CoeStar}$$

The reduction rules for the dependent type-formers $\prod$ and $\sum$ are quite complex, so we first consider their non-dependent counterparts for the sake of clarity. Let us first consider coercion for simple functions. Suppose we have some domain and range $A, B : \mathbf{I} \to *$ and a function $f : A\ \mathbf{i}_0 \to B\ \mathbf{i}_0$, we want to coerce it to a function of type $A\ \mathbf{i}_1 \to B\ \mathbf{i}_1$. We first λ-abstract to bring a variable $x : A\ \mathbf{i}_1$ in scope. We use negation and coercion to cast it to $A\ \mathbf{i}_0$:

$$[i.A\ (-i)] \triangleright x : A\ \mathbf{i}_0$$

We then apply f to the result, and cast the application from $B\ \mathbf{i}_0$ to $B\ \mathbf{i}_1$. As such the following coercion rule for simply-typed functions is admissible:

$$\frac{}{[i.A \to B] \triangleright f \Rightarrow \lambda x.[i.B] \triangleright (f\ ([i.A[i := -i]] \triangleright x))}$$

Coercion of dependent functions is a bit more complicated. We have a range $B : \prod i : \mathbf{I}.A\ i \to *$ and a function $f : \prod x : A\ \mathbf{i}_0.B\ \mathbf{i}_0\ x$, so the result of the application has to be cast from $B\ \mathbf{i}_0\ ([\lambda i.A\ (-i).x]\triangleright)$ to $B\ \mathbf{i}_1\ x$. Here, we use the maximum operator ($\vee$) to get a type-correct result. The full reduction behaviour for coercion on dependent functions is given by the rule:

$$\frac{g := \lambda j.\lambda x.[i.A[i := j \vee -i]] \triangleright x}{[i.\prod x : A.B] \triangleright f \Rightarrow \lambda x.[j.B[i := j, x := g\ j\ x]] \triangleright f\ (g\ \mathbf{i}_0\ x)}\ \textsc{CoePi}$$

Note that we define some abbreviations above the line to keep the rule concise.

Now, we want to define coercion for dependent pairs. Let us first consider how coercion would go for simple pairs. We once again have a domain and range $A, B : \mathbf{I} \to *$ and now a term $t : A\ \mathbf{i}_0 \times B\ \mathbf{i}_0$. There is no need for symmetry this time, as we can just coerce the pair component-wise. The following rule is admissible:

$$\frac{}{[i.A \times B] \triangleright t \Rightarrow \langle [i.A] \triangleright t.\pi_1, [i.B] \triangleright t.\pi_2 \rangle}$$

For dependent pairs, we once again need some additional effort to correct the types, this time using the minimum ($\wedge$) operator:

$$\frac{g := \lambda j.\lambda t.[i.A[i := j \wedge i]] \triangleright t}{[i.\sum x : A.B] \triangleright t \Rightarrow \langle g\ \mathbf{i}_1\ (t.\pi_1), [j.B[i := j, x := g\ j\ (t.\pi_1)]] \triangleright t.\pi_2 \rangle}\ \textsc{CoeSigma}$$

There are no reduction rules for coercion of equality proofs, but we do not need any. Equality proofs are computationally irrelevant, as no reduction rule depends on their structure. In particular, the equality eliminator does not inspect its proof. We reify this fact by adding UIP as a conversion rule.

$$\frac{\Gamma \vdash e, e' : t =_A u}{\Gamma \vdash e \equiv e' : t =_A u}\ \text{UIP}$$

Definition 2. We define Truncated Type Theory (TTT) as the extension of the Calculus of Constructions (CC) with the $\sum$-type; the interval with De Morgan operations; an equality type with definitional UIP; and a coercion operator with type-directed reduction rules.

Chapter 5

Metatheory

There are a number of interesting meta-theoretical properties that we would like a type-system to have. An extensive treatment of the meta-theory of type theory can be found in [8]. We discuss a number of them in relation to TTT.

5.1 Subject Reduction

Definition 3. A type system has subject reduction if the following rule is admissible.

$$\frac{\Gamma \vdash t : A \qquad t \Rightarrow u}{\Gamma \vdash u : A}$$

In other words, reduction preserves typing. This is intuitively important for correctness, as we would not expect a program of one type to yield a term of a different type after normalization. All new reduction rules in TTT preserve typing. It is worth noting that the reduction rules for coercion on $\prod$ and $\sum$ types are simplified with respect to Cubical Agda's, the addition of the CoeId rule ensures type preservation in these cases.

Conjecture 1. TTT has Subject Reduction.

5.2 Strong Normalization

A type system is said to be Strongly Normalizing (SN) if for any well-typed term, repeated application of reduction rules will eventually yield a term for which no reduction rules apply. A term that is not reducible is said to be in normal form. In other words, in a strongly normalizing type system all computations are terminating. SN for a type theory is a desirable property because, together with Confluence, it implies decidability of type-checking. To check if two types are convertible, simply reduce both and compare their normal forms.

CC is known to be strongly normalizing [16], though extensions with proof-irrelevant or observational equality have been shown to break SN [1, 28]. The proof-irrelevant equality in TTT is restricted however, to only equate objects, not constructors or kinds. One consequence is that the counterexample to SN, given by Abel and Coquand [1], is not derivable in TTT. In general, we conjecture that the extensions of TTT preserve SN.

Conjecture 2. TTT is Strongly Normalizing.

5.3 Confluence

Definition 4. Let $t \Rightarrow' u$ denote the reflexive and transitive closure of the reduction relation $t \Rightarrow u$. Let $t_1, t_2, t_3, t_4 \in \text{Term}$. A type system is confluent if $t_1 \Rightarrow' t_2$ and $t_1 \Rightarrow' t_3$ and $t_2 \Rightarrow' t_4$ implies $t_3 \Rightarrow' t_4$.

With SN, this implies reduction to be deterministic. CC is well known to be confluent [7]. The reduction rules introduced by TTT introduce no new critical pairs, so confluence should be preserved.

Conjecture 3. TTT is confluent.

5.4 Consistency

A type theory is consistent if it is not possible to prove all theorems. This is quite important if a type system is to serve as a proof assistant. For impredicative type theories, this means that the type $\prod A : *.A$ is not inhabited in the empty context. We conjecture that TTT is consistent.

Conjecture 4. TTT is consistent.

An interesting property of CC is that it admits both proof-relevant and proof-irrelevant models. A proof-irrelevant interpretation of an impredicative sort sees it purely as a deductive system. Types formed in $*$ are seen as mere propositions, and their inhabitants are considered equal. On the other hand, a proof-relevant interpretation sees the inhabitants of types in $*$ as programs, which are distinguishable. The notion of proof-irrelevance can be encoded as a proposition in TTT (or in CC, using Leibniz equality instead):

$$\prod A : *. \prod x : A. \prod y : A. x = y$$

The possibility of both models is reflected by the fact that neither proof-irrelevance nor its negation are derivable. Adding certain axioms or extensions may imply one model or the other. For example, the law of the excluded middle implies proof-irrelevance, whereas a type of booleans that can eliminate into $*$ (compute types) implies the opposite. We conjecture that TTT preserves this property, despite its extensions with respect to CC.

Conjecture 5. Neither proof-irrelevance nor its negation are derivable in TTT.

That being said, TTT is only really interesting with a proof-relevant model in mind. New theorems derivable in TTT, such as Function Extensionality, follow trivially from proof-irrelevance, as it does for Leibniz equality in CC.

5.5 Canonicity

A term of a certain type is said to be canonical if it is built from the constructors of that type. A type system is said to enjoy canonicity if any term reduces to canonical form in the empty context. This basically means that terms compute, and the system can serve as a foundation for a programming language. The exact definition of Canonicity differs per type system, as it depends on the type formers of a system. The terms of TTT are all convertible with introduction forms due to their types satisfying η-laws, so a more consequential definition of canonicity can be given with regards to Church-encoded booleans:

Definition 5. TTT satisfies canonicity if for every term t in the empty context, if $\cdot \vdash t : \prod X : *.X \to X \to X$, then $t \equiv \lambda X.\lambda x.\lambda y.x$ or $t \equiv \lambda X.\lambda x.\lambda y.y$

Adding axioms to a type system can break canonicity, as axioms can block computation. We remark that adding an extensional equality type to CC, while preserving canonicity, is the goal of TTT.

Conjecture 6. TTT enjoys canonicity.

While a proof for canonicity is out of scope for this thesis, we can analyse the computational behaviour of TTT, in particular about coercions. The following conjectures are used in the evaluation algorithm in Section 6. Let $\Gamma \in \text{Context}$, $A, B, t, u \in \text{Term}$.

Conjecture 7. Suppose $\Gamma \vdash ([i.A] \triangleright t)\ u : B$, then $([i.A] \triangleright t)\ u$ is a redex.

Conjecture 8. Suppose $\Gamma \vdash ([i.A] \triangleright t).\pi_1 : B$, then $([i.A] \triangleright t).\pi_1$ is a redex.

Conjecture 9. Suppose $\Gamma \vdash ([i.A] \triangleright t).\pi_2 : B$, then $([i.A] \triangleright t).\pi_2$ is a redex.

Conjecture 10. Suppose $\Gamma \vdash A : \square$, then $([i.A] \triangleright t)$ is a redex.

Proof Sketch By induction on the normal form of A. Note that $A[i := \mathbf{i}_1]$ must be convertible with a type- or kind-former in order for any of the conjectured redexes to be well-typed. If A is a type- or kind-former, it is a redex according to the CoePi, CoeSigma or CoeStar rules. Any other term is not well typed. We remark in particular that the equality eliminator $\mathbf{Elim}_=$ cannot be a type-constructor, because equality in TTT only equates objects, not constructors or kinds.

Chapter 6

Type Checking

In this section we describe a Bidirectional Type Checking algorithm for TTT, based on an algorithm by Coquand [13]. The algorithm accepts a slightly modified presentation of the syntax, which we describe in Section 6.1. Terms may be reduced to normal form by the evaluation algorithm described in Section 6.2. Convertibility of normal forms is decided by a type-directed conversion checker described in Section 6.3. Finally, we present a bidirectional type-checking algorithm in Section 6.4.

6.1 Syntax

We have terms (T) defined the grammar below, which is the input for the type-checking algorithm. Variables are natural numbers, denoted I in the grammar

$$\begin{aligned} \mathrm{T} ::= \star \mid \square \mid \square_{\mathbf{I}} \mid \Pi \mathrm{T}.\mathrm{T} \mid \Sigma \mathrm{T}.\mathrm{T} \mid \lambda \mathrm{T} \mid \langle \mathrm{T}, \mathrm{T} \rangle \mid \mathrm{I} \mid \mathrm{T}\ \mathrm{T} \mid \mathrm{T}.\pi_1 \mid \mathrm{T}.\pi_2 \\ \mid \mathbf{i}_0 \mid \mathbf{i}_1 \mid \mathbf{I} \mid -\mathrm{T} \mid \mathrm{T} \vee \mathrm{T} \mid \mathrm{T} \wedge \mathrm{T} \mid \mathbf{Eq}(\mathrm{T}, \mathrm{T}, \mathrm{T}) \\ \mid \mathbf{Intro}_{=}(\mathrm{T}, \mathrm{T}) \mid \mathrm{T} \triangleright \mathrm{T} \mid \mathbf{Elim}_{=}(\mathrm{T}, \mathrm{T}, \mathrm{T}, \mathrm{T}, \mathrm{T}) \mid \mathbf{let}\ \mathrm{T} : \mathrm{T}\ \mathbf{in}\ \mathrm{T} \end{aligned}$$

The syntax differs from that presented in Chapter 4 in three respects: For one, instead of named variables we use De Bruijn indices (I), represented by natural numbers. This representation simplifies evaluation of terms, albeit at the cost of readability. Second difference is the addition of a let-in construct. The type-checking algorithm rejects terms that contain β-redexes. The let-in construct recovers some of the expressivity lost due to this fact. Thirdly, coercion no longer binds an interval variable in the type, instead expecting a function out of the interval. This is to simplify the presentation.

Terms are evaluated to normal forms (NF), which are defined by the following grammar. Here variables are natural numbers with a type annotation, denoted L : Nf in the grammar.

$$\mathrm{Ne} ::= \mathrm{L} : \mathrm{Nf} \mid \mathbf{Elim}_{=}(\mathrm{Nf}, \mathrm{Nf}, \mathrm{Nf}, \mathrm{Nf}, \mathrm{L}) \mid \mathbf{D}(\mathrm{Nf}) \mid \mathbf{R}(\mathrm{Nf}) \mid \mathrm{Ne}.\pi_1 \mid \mathrm{Ne}.\pi_2 \mid \mathrm{Ne}\ \mathrm{Nf}$$

$$\begin{aligned}\mathrm{Nf} ::= \star \mid \square \mid \square_{\mathbf{I}} \mid \Pi\mathrm{Nf}.\mathrm{T}[\sigma] \mid \Sigma\mathrm{Nf}.\mathrm{T}[\sigma] \mid \lambda\mathrm{T}[\sigma] \mid \langle \mathrm{Nf}, \mathrm{Nf}\rangle \mid \mathrm{Ne} \\ \mid \mathbf{i}_0 \mid \mathbf{i}_1 \mid \mathbf{I} \mid -\mathrm{Nf} \mid \mathrm{Nf} \vee \mathrm{Nf} \mid \mathrm{Nf} \wedge \mathrm{Nf} \mid \mathrm{Nf} \triangleright \mathrm{Nf} \\ \mid \mathbf{Eq}(\mathrm{Nf}, \mathrm{Nf}, \mathrm{Nf}) \mid \mathbf{Intro}_{=}(\mathrm{Nf}, \mathrm{Nf})\end{aligned}$$

There are a number of notable aspects of normal forms:

1. A Nf is either head-normal or a neutral (Ne). A neutral term is either a variable, or an elimination (application, projection, equality elimination) stuck on a neutral.

2. Variables are represented by de Bruijn levels (L), the counterpart to de Bruijn indices. Furthermore, they are annotated with their type, which is used in the conversion algorithm.

3. $\mathbf{D}(-)$ and $\mathbf{R}(-)$ are dummies used in evaluation and are explained in Section 6.2

4. Substitutions (σ) are used in evaluation, as is explained in Section 6.2.

5. The domains of binders (Π,Σ,λ) are not normal forms but *closures*, consisting of a term and a substitution, denoted as $T[\sigma]$ These will be explained in Section 6.2.

   Finally, we have substitutions σ for evaluation, and contexts Γ for type checking. Both are lists of normal forms.

$$\Gamma ::= \cdot \mid \Gamma, \mathrm{Nf} \qquad\qquad \sigma ::= \cdot \mid \sigma, \mathrm{Nf}$$

Note that variables in contexts are unnamed, which is consistent with the nameless presentation of the syntax.

6.2 Evaluation

The evaluation algorithm uses Normalization by Evaluation (NbE). The first description of NbE for typed lambda calculi was given in [10]. Our algorithm is mainly based on [22]. We extend it to handle the cubical constructs of TTT, With a novel way to compute coercions. Evaluation is carried out by the mutually recursive functions:

$$\begin{aligned}[\![-]\!][-] &: \mathrm{T} \to \sigma \to \mathrm{Nf} \\ - \cdot - &: \mathrm{Nf} \to \mathrm{Nf} \to \mathrm{Nf} \\ \mathbf{fst}(-) &: \mathrm{Nf} \to \mathrm{Nf} \\ \mathbf{snd}(-) &: \mathrm{Nf} \to \mathrm{Nf}\end{aligned}$$

$$
\begin{aligned}
\mathbf{neg}(-) &: \mathrm{Nf} \to \mathrm{Nf} \\
\mathbf{max}(-,-) &: \mathrm{Nf} \to \mathrm{Nf} \to \mathrm{Nf} \\
\mathbf{min}(-,-) &: \mathrm{Nf} \to \mathrm{Nf} \to \mathrm{Nf}
\end{aligned}
$$

$$
\begin{aligned}
[\![\star]\!][\sigma] &= \star \\
[\![\Box]\!][\sigma] &= \Box \\
[\![\Box_{\mathbf{I}}]\!][\sigma] &= \Box_{\mathbf{I}} \\
[\![\Pi t.u]\!][\sigma] &= \Pi([\![t]\!][\sigma]).u[\sigma] \\
[\![\Sigma t.u]\!][\sigma] &= \Sigma([\![t]\!][\sigma]).u[\sigma] \\
[\![\lambda t]\!][\sigma] &= \lambda t[\sigma] \\
[\![\langle t, u\rangle]\!][\sigma] &= \langle [\![t]\!][\sigma], [\![u]\!][\sigma]\rangle \\
[\![\mathbf{let}\ t : \_\ \mathbf{in}\ u]\!][\sigma] &= [\![u]\!][(\sigma, [\![t]\!][\sigma])] \\
[\![t\ u]\!][\sigma] &= [\![t]\!][\sigma] \cdot [\![u]\!][\sigma] \\
[\![t.\pi_1]\!][\sigma] &= \mathbf{fst}([\![t]\!][\sigma]) \\
[\![t.\pi_2]\!][\sigma] &= \mathbf{snd}([\![t]\!][\sigma]) \\
[\![\mathbf{i}_0]\!][\sigma] &= \mathbf{i}_0 \\
[\![\mathbf{i}_1]\!][\sigma] &= \mathbf{i}_1 \\
[\![\mathbf{I}]\!][\sigma] &= \mathbf{I} \\
[\![-i]\!][\sigma] &= \mathbf{neg}([\![i]\!][\sigma]) \\
[\![i \wedge j]\!][\sigma] &= \mathbf{min}([\![i]\!][\sigma], [\![j]\!][\sigma]) \\
[\![i \vee j]\!][\sigma] &= \mathbf{max}([\![i]\!][\sigma], [\![j]\!][\sigma]) \\
[\![t \triangleright u]\!][\sigma] &= [\![t]\!][\sigma] \triangleright [\![u]\!][\sigma] \\
[\![\mathbf{Eq}(a,t,u)]\!][\sigma] &= \mathbf{Eq}([\![a]\!][\sigma], [\![t]\!][\sigma], [\![u]\!][\sigma]) \\
[\![\mathbf{Intro}_{=}(a,t)]\!][\sigma] &= \mathbf{Intro}_{=}([\![a]\!][\sigma], [\![t]\!][\sigma]) \\
[\![\mathbf{Elim}_{=}(a,t,u,e,i)]\!][\sigma] &= [\![t]\!][\sigma]\ \mathbf{if}\ [\![i]\!][\sigma] = \mathbf{i}_0 \\
[\![\mathbf{Elim}_{=}(a,t,u,e,i)]\!][\sigma] &= [\![u]\!][\sigma]\ \mathbf{if}\ [\![i]\!][\sigma] = \mathbf{i}_1 \\
[\![\mathbf{Elim}_{=}(a,t,u,e,i)]\!][\sigma] &= \mathbf{Elim}_{=}([\![a]\!][\sigma], [\![t]\!][\sigma], [\![u]\!][\sigma], [\![e]\!][\sigma], [\![i]\!][\sigma])\ \mathbf{otherwise} \\
[\![0]\!][\sigma, v] &= v \\
[\![i+1]\!][\sigma, v] &= [\![i]\!][\sigma]
\end{aligned}
$$

$$
\begin{aligned}
\mathbf{neg}(\mathbf{i}_0) &= \mathbf{i}_1 \\
\mathbf{neg}(\mathbf{i}_1) &= \mathbf{i}_0 \\
\mathbf{neg}(-i) &= i \\
\mathbf{neg}(i \wedge j) &= \mathbf{neg}(i) \vee \mathbf{neg}(j) \\
\mathbf{neg}(i \vee j) &= \mathbf{neg}(i) \wedge \mathbf{neg}(j) \\
\mathbf{neg}(i) &= -i
\end{aligned}
$$

$$\begin{aligned}
\mathbf{min}(\mathbf{i}_0, i) &= \mathbf{i}_0 & \mathbf{max}(\mathbf{i}_0, i) &= i \\
\mathbf{min}(\mathbf{i}_1, i) &= i & \mathbf{max}(\mathbf{i}_1, i) &= \mathbf{i}_1 \\
\mathbf{min}(i, \mathbf{i}_0) &= \mathbf{i}_0 & \mathbf{max}(i, \mathbf{i}_0) &= i \\
\mathbf{min}(i, \mathbf{i}_1) &= i & \mathbf{max}(i, \mathbf{i}_1) &= \mathbf{i}_1 \\
\mathbf{min}(i, j) &= i \wedge j & \mathbf{max}(i, j) &= i \vee j
\end{aligned}$$

$$\begin{aligned}
\mathbf{aux} &= \lambda(\lambda\lambda\lambda\lambda(3\ 0\ (2\ 0\ 1)))[\cdot] && (= \lambda abgxj.b\ j\ (g\ j\ x)) \\
\mathbf{aux}_\Pi &= \lambda(\lambda\lambda(\lambda 3(2 \vee -0)) \triangleright 0)[\cdot] && (= \lambda ajx.(\lambda i.a\ (-i \vee j)) \triangleright x) \\
\mathbf{aux}_\Sigma &= \lambda(\lambda\lambda(\lambda 3(2 \wedge 0)) \triangleright 0)[\cdot] && (= \lambda ajt.(\lambda i.a\ (i \wedge j)) \triangleright t)
\end{aligned}$$

$$\begin{aligned}
\lambda t[\sigma] \cdot v &= [\![t]\!][\sigma, v] \\
(a \triangleright f) \cdot v &= (\mathbf{aux} \cdot \mathbf{D}(a) \cdot \mathbf{R}(a) \cdot (\mathbf{aux}_\Pi \cdot \mathbf{D}(a)) \cdot v) \triangleright (f \cdot (\mathbf{aux}_\Pi \cdot \mathbf{D}(a) \cdot \mathbf{i}_0 \cdot v)) \\
\mathbf{D}(v) \cdot v' &= a \ \mathbf{if}\ v \cdot v' = \Pi a.b[\sigma] \\
\mathbf{D}(v) \cdot v' &= a \ \mathbf{if}\ v \cdot v' = \Sigma a.b[\sigma] \\
\mathbf{R}(v) \cdot v' &= \lambda b[\sigma] \ \mathbf{if}\ v \cdot v' = \Pi a.b[\sigma] \\
\mathbf{R}(v) \cdot v' &= \lambda b[\sigma] \ \mathbf{if}\ v \cdot v' = \Sigma a.b[\sigma] \\
v \cdot v' &= v\ v'
\end{aligned}$$

$$\begin{aligned}
\mathbf{fst}(\langle v, w \rangle) &= v \\
\mathbf{fst}((a \triangleright v)) &= \mathbf{D}(a) \triangleright \mathbf{fst}(t) \\
\mathbf{fst}(v) &= v.\pi_1 \\
\mathbf{snd}(\langle v, w \rangle) &= u \\
\mathbf{snd}((a \triangleright v)) &= (\mathbf{aux} \cdot \mathbf{D}(a) \cdot \mathbf{R}(a) \cdot (\mathbf{aux}_\Sigma \cdot \mathbf{D}(a)) \cdot (\mathbf{fst}(t))) \triangleright (\mathbf{snd}(t)) \\
\mathbf{snd}(v) &= v.\pi_2
\end{aligned}$$

$[\![t]\!][\sigma]$ means to evaluate the term t under the substitution σ. The procedure has as precondition that the range of free indices in t is smaller than the length of σ. The evaluation of an index n under the substitution σ is simply the nth value of σ, so the precondition ensures that all free variables in t will be assigned a value by σ.

Due to this precondition, evaluation does not proceed under binders, instead creating a closure, consisting of the term to be evaluated, and the substitution. Evaluation may proceed when there is a value assignment for the bound variable, such as when a value is applied to a λ-closure.

The functions $- \cdot -$, **fst**, **snd**, **neg**, **min** and **max** just reconstruct applications, projections and de Morgan operations, respectively. β-redexes are reduced as they emerge.

Application of a λ-closure $\lambda t[\sigma]$ and a value v resumes evaluation of t, with the substitution σ extended by v.

The treatment of coercion in our system is of particular interest. The coercions make use of some auxiliary abbreviations. We added named versions for clarity. Let us first remark that coercions only compute under application or projections. This corresponds to the following reduction rules, which are admissible because of η-equivalence.

$$\frac{\begin{array}{c}A' := \lambda i.A \qquad B' := \lambda i.\lambda x.B \\ g := \lambda j.\lambda x.(\lambda i.A'(j \vee -i)) \triangleright x\end{array}}{\begin{array}{c}((\lambda i.\prod x : A.B) \triangleright f)\ t \Rightarrow \\ (\lambda j.B'\ j\ (g\ j\ t)) \triangleright f\ (g\ \mathbf{i}_0\ t)\end{array}}$$

$$\frac{\begin{array}{c}A' := \lambda i.A \qquad B' := \lambda i.\lambda x.B \\ g := \lambda j.\lambda t.(\lambda i.A'\ (j \wedge i)) \triangleright t\end{array}}{((\lambda i.\sum x : A.B) \triangleright t).\pi_1 \Rightarrow g\ \mathbf{i}_1\ (t.\pi_1)}$$

$$\frac{\begin{array}{c}A' := \lambda i.A \qquad B' := \lambda i.\lambda x.B \\ g := \lambda j.\lambda t.(\lambda i.A'\ (j \wedge i)) \triangleright t\end{array}}{\begin{array}{c}((\lambda i.\sum x : A.B) \triangleright t).\pi_2 \Rightarrow \\ (\lambda j.B'\ j\ (g\ j\ (t.\pi_1))) \triangleright t.\pi_2\end{array}}$$

The redexes defined by the reduction rules have the type former under a lambda abstraction binding an interval variable. This would present a problem for NbE, because the algorithm does not reduce under binders, it cannot check if a term under a λ-abstraction normalizes to a type former. Fortunately, in our theory, well-typed coercions under applications and projections are guaranteed to be redexes, as stated in Chapter 5. This is why we introduce the domain- and range extraction closures, $\mathbf{D}(-)$ and $\mathbf{R}(-)$. We illustrate their reduction behaviour with pseudo-rules. The extraction closures only reduce under application. They await a second argument, which they apply to the type function. Then they extract the components.

$$\frac{}{\mathbf{D}(\lambda j.\prod x : A.B)\ i \Rightarrow A[j := i]}$$

$$\frac{}{\mathbf{R}(\lambda j.\prod x : A.B)\ i \Rightarrow \lambda x.B[j := i]}$$

6.3 Conversion checking

$\beta\eta$-conversion is decided for normal forms by the conversion checker. The conversion checking algorithm is type-directed, similarly to the algorithm presented in [26]. Since it is a fallible algorithm, and we do not want to clutter the presentation with failure propagation, we opt to present the

algorithm as syntax-directed derivation rules. We remark that that the algorithm assumes the input terms are well-typed, as a precondition. There are three judgments, which correspond to mutually recursive functions:

$\Gamma \vdash v \equiv w : a$ In context Γ, v and u of type a are convertible (type-directed)

$\Gamma \vdash v \equiv w$ In context Γ, v and u are convertible (untyped)

$\Gamma \vdash v \equiv w \rightsquigarrow a$ In context Γ, v and u are convertible and synthesise type a

We present and explain the conversion rules below. Type-directed conversion eagerly η-expands terms. There are three interesting cases. In the case for dependent functions, both functions are applied to a fresh unknown (De Bruijn level). The function $\mathbf{fresh}(\Gamma)$ just returns the length of Γ.

$$\frac{l := \mathbf{fresh}(\Gamma) \qquad \Gamma, a \vdash v \cdot (l : a) \equiv v' \cdot (l : a) : [\![b]\!][\sigma, (l : a)]}{\Gamma \vdash v \equiv v' : \Pi a.b[\sigma]}$$

Pairs are compared component-wise:

$$\frac{\Gamma \vdash \mathbf{fst}(v) \equiv \mathbf{fst}(v') : a \qquad \Gamma \vdash \mathbf{snd}(v) \equiv \mathbf{snd}(v') : [\![b]\!][\sigma, \mathbf{fst}(v)]}{\Gamma \vdash v \equiv v' : \Sigma a.b[\sigma]}$$

Equality proofs are unconditionally convertible:

$$\frac{}{\Gamma \vdash e \equiv e' : t =_a u}$$

For neutral types and sorts, no η-rules apply, so the algorithm defers to untyped conversion:

$$\frac{a \in \mathrm{Ne} \vee a \in \mathcal{S} \qquad \Gamma \vdash v \equiv v'}{\Gamma \vdash v \equiv v' : a}$$

The untyped conversion rules mostly deal with congruences, which we omit here. Interesting exceptions are coercions, binders, binary de Morgan operations and neutrals. The rule for coercions implements the CoeId rule with transitivity, in a syntax-directed manner.

$$\frac{\Gamma \vdash a \cdot \mathbf{i}_0 \equiv a \cdot \mathbf{i}_1 \qquad \Gamma \vdash t \equiv t'}{\Gamma \vdash a \triangleright t \equiv t'}$$

For binders, a fresh unknown value is generated to evaluate the range closures, which are then compared.

$$\frac{l := \mathbf{fresh}(\Gamma) \qquad \Gamma \vdash a \equiv a' \qquad \Gamma, a \vdash [\![b]\!][\sigma, (l : a)] \equiv [\![b']\!][\sigma', (l : a)]}{\Gamma \vdash \Pi a.b[\sigma] \equiv \Pi a'.b'[\sigma']}$$

Neutral terms are deferred to the type-synthesizing conversion check, though the resulting type will be discarded.

$$\frac{v, v' \in \mathrm{Ne} \qquad \Gamma \vdash v \equiv v' \rightsquigarrow a}{\Gamma \vdash v \equiv v'}$$

For the binary de Morgan operations, the algorithm checks for congruence or commutativity in order.

$$\frac{\Gamma \vdash i \equiv i' \qquad \Gamma \vdash j \equiv j'}{\Gamma \vdash i \wedge j \equiv i' \wedge j'}$$

$$\frac{\Gamma \vdash i \equiv j' \qquad \Gamma \vdash j \equiv i'}{\Gamma \vdash i \wedge j \equiv i' \wedge j'}$$

For neutral terms, conversion checking is also untyped, but the algorithm synthesizes the type if successful. The synthesized type is used for the type-directed check of application arguments.

$$\frac{}{\Gamma \vdash (l : a) \equiv (l : a) \rightsquigarrow a}$$

$$\frac{\Gamma \vdash a \equiv a' : \mathbf{I} \to \star \qquad \Gamma \vdash t \equiv t' : a \cdot \mathbf{i}_0 \qquad \Gamma \vdash u \equiv u' : a \cdot \mathbf{i}_1 \qquad \Gamma \vdash i \equiv i'}{\Gamma \vdash \mathbf{Elim}_{=}(a, t, u, e, i) \equiv \mathbf{Elim}_{=}(a', t', u', e', i') \rightsquigarrow a \cdot i}$$

$$\frac{\Gamma \vdash f \equiv f' \rightsquigarrow \Pi a.b[\sigma] \qquad \Gamma \vdash v \equiv v' : a}{\Gamma \vdash f\ v \equiv f'\ v' \rightsquigarrow [\![b]\!][\sigma, v]}$$

$$\frac{\Gamma \vdash t \equiv t' \rightsquigarrow \Sigma a.b[\sigma]}{\Gamma \vdash t.\pi_1 \equiv t'.\pi_1 : a}$$

$$\frac{\Gamma \vdash t \equiv t' \rightsquigarrow \Sigma a.b[\sigma]}{\Gamma \vdash t.\pi_2 \equiv t'.\pi_2 : [\![b]\!][\sigma, t.\pi_1]}$$

6.4 Bidirectional type-checking

The type-checking algorithm is based on Kovács work [22], which in turn is based on an algorithm by Coquand [13]. We present the algorithm again as syntax-directed derivation rules. There are two judgments:

$$\Gamma, \sigma \vdash t :_? v \Downarrow w$$
$$\Gamma, \sigma \vdash t \Uparrow u : v$$

We read $\Gamma, \sigma \vdash t :_? v \Downarrow w$ as: "In context Γ, under substitution σ, the term t checks against the type v, and evaluates to w". We read $\Gamma, \sigma \vdash t \Uparrow u : v$ as: "In context Γ, under substitution σ, the term t synthesizes the type v and evaluates to w. The fact that the checking and synthesis algorithms

normalize well-typed terms is new with respect to Coquands algorithm, but the other than that our presentation is standard.

Type assignment works for let-expressions, lambda-abstractions and pairs. Any other case defers to the Conv rule. Note that there are no synthesis rules for let, lambda and pairs. Furthermore, applications can only be typed if the function type can be synthesized, similarly for projections of pairs. As such, the algorithm will fail to type terms containing β-redexes.

$$\frac{}{\Gamma, \sigma \vdash * \Uparrow * : \Box}\ \textsc{Star}$$

$$\frac{}{\Gamma, \sigma \vdash \mathbf{I} \Uparrow \mathbf{I} : \Box_{\mathbf{I}}}\ \mathrm{I} \qquad \frac{}{\Gamma, \sigma \vdash \mathbf{i}_0 \Uparrow \mathbf{i}_0 : \mathbf{I}}\ \mathrm{I0} \qquad \frac{}{\Gamma, \sigma \vdash \mathbf{i}_1 \Uparrow \mathbf{i}_1 : \mathbf{I}}\ \mathrm{I1}$$

$$\frac{\Gamma, \sigma \vdash i :_? \mathbf{I} \Downarrow i'}{\Gamma, \sigma \vdash -i \Uparrow \mathbf{neg}(i') : \mathbf{I}}\ \textsc{Neg}$$

$$\frac{\Gamma, \sigma \vdash i :_? \mathbf{I} \Downarrow i' \qquad \Gamma, \sigma \vdash j :_? \mathbf{I} \Downarrow j'}{\Gamma, \sigma \vdash i \wedge j \Uparrow \mathbf{min}(i', j') : \mathbf{I}}\ \textsc{Min}$$

$$\frac{\Gamma, \sigma \vdash i :_? \mathbf{I} \Downarrow i' \qquad \Gamma, \sigma \vdash j :_? \mathbf{I} \Downarrow j'}{\Gamma, \sigma \vdash i \vee j \Uparrow \mathbf{max}(i', j') : \mathbf{I}}\ \textsc{Max}$$

$$\frac{}{(\Gamma, v), (\sigma, w) \vdash 0 \Uparrow v : w}\ \mathrm{IX}_0 \qquad \frac{\Gamma, \sigma \vdash i \Uparrow v : w}{(\Gamma, v), (\sigma, w) \vdash Si \Uparrow v : w}\ \mathrm{IX}_S$$

$$\frac{l := \mathbf{fresh}(\Gamma) \qquad (\Gamma, a), (\sigma, l) \vdash t :_? (\llbracket b \rrbracket[(\sigma', l)]) \Downarrow v}{\Gamma, \sigma \vdash \lambda t :_? \Pi a.b[\sigma'] \Downarrow \lambda t[\sigma]}\ \textsc{Lambda}$$

$$\frac{\Gamma, \sigma \vdash t \Uparrow v : \Pi a.b[\sigma'] \qquad \Gamma, \sigma \vdash u :_? a \Downarrow w}{\Gamma, \sigma \vdash t\ u \Uparrow v \cdot w : \llbracket b \rrbracket[(\sigma', w)]}\ \textsc{App}$$

$$\frac{\Gamma, \sigma \vdash t \Uparrow v : \Sigma a.b[\sigma']}{\Gamma, \sigma \vdash t.\pi_1 \Uparrow \mathbf{fst}(v) : a}\ \textsc{Fst}$$

$$\frac{\Gamma, \sigma \vdash t \Uparrow v : \Sigma a.b[\sigma']}{\Gamma, \sigma \vdash t.\pi_2 \Uparrow \mathbf{snd}(v) : \llbracket b \rrbracket[(\sigma, (v \cdot \pi_0))]}\ \textsc{Snd}$$

$$\frac{\Gamma, \sigma \vdash t \Uparrow v : w \qquad \Gamma, \sigma \vdash w \equiv w' \qquad v \in \mathrm{Ne}}{\Gamma, \sigma \vdash t :_? v \Downarrow w'}\ \textsc{Conv}$$

$$\frac{l := \mathbf{fresh}(\Gamma) \qquad \Gamma, \sigma \vdash a \Uparrow v : s_0 \qquad (\Gamma, a), (\sigma, l) \vdash b \Uparrow w : s_1 \qquad (s_0, s_1) \in \mathcal{R}}{\Gamma, \sigma \vdash \Pi a.b \Uparrow \Pi v.b[\sigma] : s_1}\ \mathrm{PI}$$

$$\frac{l := \mathbf{fresh}(\Gamma) \qquad \Gamma, \sigma \vdash a \Uparrow v : * \qquad (\Gamma, v), (\sigma, l) \vdash b \Uparrow w : *}{\Gamma, \sigma \vdash \Sigma a.b \Uparrow \Sigma v.w[\sigma] : *}\ \textsc{Sigma}$$

$$\frac{\Gamma, \sigma \vdash t :_? v \Downarrow a \qquad \Gamma, \sigma \vdash u :_? w \Downarrow [\![b]\!][(S', v)]}{\Gamma, \sigma \vdash \langle t, u\rangle :_? \Sigma a.b[\sigma'] \Downarrow \langle v, w\rangle}\ \textsc{Pair}$$

$$\frac{\Gamma, \sigma \vdash a \Uparrow a' : s \qquad s \in \mathcal{S} \qquad \Gamma, \sigma \vdash t :_? a' \Downarrow v \qquad (\Gamma, a'), (\sigma, v) \vdash u :_? b \Downarrow \_}{\Gamma, \sigma \vdash \mathbf{let}\ t : a\ \mathbf{in}\ u :_? b \Downarrow [\![u]\!][\sigma, v]}\ \textsc{Let}$$

$$\frac{\Gamma, \sigma \vdash a :_? \Pi\mathbf{I}.\, * [\cdot] \Downarrow a' \qquad \Gamma, \sigma \vdash t :_? a' \cdot \mathbf{i}_0 \Downarrow t' \qquad \Gamma, \sigma \vdash u :_? a' \cdot \mathbf{i}_1 \Downarrow u'}{\Gamma, \sigma \vdash \mathbf{Eq}(a, t, u) \Uparrow \mathbf{Eq}(a', t', u') : *}\ \textsc{Eq}$$

$$\frac{\Gamma, \sigma \vdash a :_? \Pi\mathbf{I}.\, * [\cdot] \Downarrow a' \qquad \Gamma, \sigma \vdash t :_? a' \cdot \mathbf{i}_0 \Downarrow t'}{\Gamma, \sigma \vdash \mathbf{Intro}_=(a, t) \Uparrow \mathbf{Intro}_=(a', t') : \mathbf{Eq}(a', t \cdot \mathbf{i}_0, t \cdot \mathbf{i}_1)}\ \textsc{EqIntro}$$

6.5 Implementation

We implemented a rudimentary proof assistant written in Haskell. While by no means as expressive as Coq or Agda, the checker accepts text files with definitions and axioms. Our system implements the type-checking algorithm, with a few additional features on top:

1. The user language has named variables, while the internal representation uses de Bruijn indices. The elaborator translates variable names to indices while type-checking. Binders still store the user-given names, so the prettyprinter can restore the named presentation for error reporting. This approach is similar to the one of used in the Matita proof assistant [3].

2. A file consists of a sequence of definitions. Each definition is in scope of all subsequent ones and may be unfolded for conversion checking. The evaluator does not unfold definitions eagerly, rather using glued evaluation as was described by Kovács [22].

3. A simple module system. A file is a module. An import statement in one file brings the definitions of another in scope. A name may be defined in multiple modules, and be disambiguated by prefixing it with the module name.

We implemented impredicative encodings of some inductive types such as booleans, natural numbers and quotients in the style of [4], which are accepted by our checker. The derivations are somewhat simplified: because UIP is built-in we do not need to encode Set with a $\sum$-type. One interesting result we found, due to the strong reduction behaviour of coercions, computation rules for the induction principles hold definitionally for apparently any (higher) inductive type without recursive point constructors.

Chapter 7

Conclusion & Future Work

We have developed Truncated Type Theory by extending CC with an interval, path types and type-directed coercion. We conjecture normalization and canonicity for TTT. Assuming these properties, we devised an efficient normalization procedure. We implemented a rudimentary proof-assistant in Haskell, with a proof-checking kernel of about 420 lines of code. Our checker accepts derivations of induction principles for a number of (higher) inductive types. We believe we have shown TTT to be an adequate type system for a proof-assistant by striking a balance between simplicity and expressive power.

7.1 Limitations

We have identified a number of limitations. First and foremost is the equality type in TTT being restricted to only equate objects. This is the price we pay for having impredicativity and definitional UIP. In addition to the lack of universes, this means TTT is probably unsuitable for reasoning on a higher level of abstraction. TTT is unlikely to be able to formalize category theory. But it should be a solid foundation for programming and proving, given that it is strictly stronger than System Fω.

Regarding the type checking algorithm, we suspect the type-directed conversion check is quite inefficient, at least when it comes to eta-expanding dependent pairs. If we have some pairs p, p' then the conversion checker will check $p.\pi_1 \equiv p'.\pi_1$ and $p.\pi_2 \equiv p'.\pi_2$ separately, even if both p and p' evaluate to neutral terms, and regardless of how expensive computing $p \equiv p'$ is. This potentially results in a lot of work duplication.

although inductive types are derivable and accepted by our proof-checker, the computation rules only hold propositionally, not definitionally. This is in stark contrast to proof assistants like Agda, Coq or Lean, which do satisfy definitional ι-rules [26, 24, 15]. A proof-assistant based on TTT would need to provide a additional automation for it to hold a candle to Agda, Coq or

Lean.

7.2 Future Work

There are a number of opportunities for improvement. Firstly, a number of meta-theoretical properties are merely conjectured. Proofs for canonicity and normalization would be invaluable for the goals of TTT. We believe the latter can be shown by erasure to Fω, similarly to existing proofs of SN for CC.

Secondly, we would like to address the lack of definitional ι-rules. A simplifier like Leans's [15] might adequately compensate. Another possibility would be to strengthen the theory further. TTT could be extended with inductive schemas like CIC. Although this would not result in a small proof-kernel, an inductive TTT would still have an advantage over CIC due to the computational coercions. Alternatively, we can address the lack of definitional ι-rules by adding equality reflection to the system. If we do so, we can recover decidability of type-checking by checking the full type derivations, or translating them to regular, intensional CC with extra axioms. This last approach would result in a much simpler kernel than TTT has, albeit at the cost of imposing the burden of translation upon the untrusted part of the proof system. It would also remedy the problem that type-directed conversion checks are inherently less efficient than untyped checks, due to the necessary reconstruction of type information to trigger η-rules.

7.3 Acknowledgements

I would like to thank Herman Geuvers and Niels van der Weide. Their knowledge, advice, and our discussions have been invaluable.

Bibliography

[1] Andreas Abel and Thierry Coquand. Failure of normalization in impredicative type theory with proof-irrelevant propositional equality. *Log. Methods Comput. Sci.*, 16(2), 2020.

[2] Thorsten Altenkirch, Conor McBride, and Wouter Swierstra. Observational equality, now! In Aaron Stump and Hongwei Xi, editors, *Proceedings of the ACM Workshop Programming Languages meets Program Verification, PLPV 2007, Freiburg, Germany, October 5, 2007*, pages 57–68. ACM, 2007.

[3] Andrea Asperti, Wilmer Ricciotti, Claudio Sacerdoti Coen, and Enrico Tassi. A compact kernel for the Calculus of Inductive Constructions. *Sadhana*, 34:71–144, 02 2009.

[4] Steve Awodey, Jonas Frey, and Sam Speight. Impredicative encodings of (higher) inductive types. In *Proceedings of the 33rd Annual ACM/IEEE Symposium on Logic in Computer Science*, LICS '18, page 76–85, New York, NY, USA, 2018. Association for Computing Machinery.

[5] Henk Barendregt. Introduction to Generalized Type Systems. *Journal of Functional Programming*, 1(2):125–154, 1991.

[6] Henk Barendregt and Herman Geuvers. Proof-assistants using dependent type systems. In John Alan Robinson and Andrei Voronkov, editors, *Handbook of Automated Reasoning (in 2 volumes)*, pages 1149–1238. Elsevier and MIT Press, 2001.

[7] Henk P. Barendregt. *The lambda calculus, its syntax and semantics*. North-Holland, Amsterdam, second, revised edition, 1984.

[8] Henk P. Barendregt. Lambda calculi with types. In D.M. Gabbai Samson Abramski and T.S.E. Maiboum, editors, *Handbook of Logic in Computer Science*. Oxford University Press, Oxford, 1992.

[9] Gilles Barthe and Morten Heine Sørensen. Domain-free Pure Type Systems. *Journal of Functional Programming*, 10(5):417–452, 2000.

[10] U. Berger and H. Schwichtenberg. An inverse of the evaluation functional for typed lambda -calculus. In *[1991] Proceedings Sixth Annual IEEE Symposium on Logic in Computer Science*, pages 203–211, 1991.

[11] Alonzo Church. A formulation of the simple theory of types. *The Journal of Symbolic Logic*, 5(2):56–68, 1940.

[12] Cyril Cohen, Thierry Coquand, Simon Huber, and Anders Mörtberg. Cubical Type Theory: A constructive interpretation of the univalence axiom. In Tarmo Uustalu, editor, *21st International Conference on Types for Proofs and Programs, TYPES 2015, May 18-21, 2015, Tallinn, Estonia*, volume 69 of *LIPIcs*, pages 5:1–5:34. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2015.

[13] Thierry Coquand. An algorithm for type-checking dependent types. *Sci. Comput. Program.*, 26(1-3):167–177, 1996.

[14] Thierry Coquand and Gérard Huet. The Calculus of Constructions. *Information and Computation*, 76(2):95–120, 1988.

[15] Leonardo de Moura, Soonho Kong, Jeremy Avigad, Floris Doorn, and Jakob Raumer. The lean theorem prover (system description). volume 9195, pages 378–388, 08 2015.

[16] Herman Geuvers. A short and flexible proof of strong normalization for the Calculus of Constructions. In P. Dybjer, B. Nordström, and J. Smith, editors, *Selected Papers 2nd Int. Workshop on Types for Proofs and Programs, TYPES'94, Båstad, Sweden, 6–10 June 1994*, volume 996, pages 14–38. Springer-Verlag, Berlin, 1995.

[17] Herman Geuvers. Introduction to type theory. In Ana Bove, Luís Soares Barbosa, Alberto Pardo, and Jorge Sousa Pinto, editors, *Language Engineering and Rigorous Software Development, International LerNet ALFA Summer School 2008, Piriapolis, Uruguay, February 24 - March 1, 2008, Revised Tutorial Lectures*, volume 5520 of *Lecture Notes in Computer Science*, pages 1–56. Springer, 2008.

[18] J.-Y. Girard, Yves Lafont, and Paul Taylor. *Proofs and types*, volume 7 of *Cambridge tracts in theoretical computer science*. Cambridge University Press, Cambridge, 1989.

[19] Jean-Yves Girard. *Interprétation fonctionelle et élimination des coupres de l'arithmétique d'ordre supérieur*. PhD thesis, Universite Paris VII, June 1972.

[20] Martin Hofmann. *Extensional concepts in intensional type theory*. PhD thesis, University of Edinburgh, 1995.

[21] William Alvin Howard. The formulae-as-types notion of construction. In Haskell Curry, Hindley B., Seldin J. Roger, and P. Jonathan, editors, *To H. B. Curry: Essays on Combinatory Logic, Lambda Calculus, and Formalism*. Academic Press, 1980.

[22] András Kovács. *smalltt*. https://github.com/AndrasKovacs/smalltt, 2023.

[23] Per Martin-Löf. An intuitionistic theory of types: Predicative part. In H.E. Rose and J.C. Shepherdson, editors, *Logic Colloquium '73*, volume 80 of *Studies in Logic and the Foundations of Mathematics*, pages 73–118. Elsevier, 1975.

[24] The Coq development team. *The Coq proof assistant reference manual*, 2004. Version 8.0.

[25] B. Nordström, K. Peterson, and J. M. Smith. *Programming in Martin-Löf's Type Theory : an introduction*. Oxford Science Publications, 1990.

[26] Ulf Norell. *Towards a practical programming language based on dependent type theory*. PhD thesis, Department of Computer Science and Engineering, Chalmers University of Technology, SE-412 96 Göteborg, Sweden, September 2007.

[27] Loïc Pujet and Nicolas Tabareau. Observational equality: Now for good. *Proc. ACM Program. Lang.*, 6(POPL):1–27, 2022.

[28] Loïc Pujet and Nicolas Tabareau. Impredicative observational equality. *Proc. ACM Program. Lang.*, 7(POPL), jan 2023.

[29] John C. Reynolds. Towards a theory of type structure. In Bernard J. Robinet, editor, *Programming Symposium, Proceedings Colloque sur la Programmation, Paris, France, April 9-11, 1974*, volume 19 of *Lecture Notes in Computer Science*, pages 408–423. Springer, 1974.

[30] Jonathan Sterling, Carlo Angiuli, and Daniel Gratzer. Cubical syntax for reflection-free extensional equality. In Herman Geuvers, editor, *4th International Conference on Formal Structures for Computation and Deduction, FSCD 2019, June 24-30, 2019, Dortmund, Germany*, volume 131 of *LIPIcs*, pages 31:1–31:25. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2019.

[31] The Univalent Foundations Program. *Homotopy Type Theory: Univalent Foundations of Mathematics*. https://homotopytypetheory.org/book, Institute for Advanced Study, 2013.

[32] Andrea Vezzosi, Anders Mörtberg, and Andreas Abel. Cubical Agda: A dependently typed programming language with univalence and higher inductive types. *J. Funct. Program.*, 31:e8, 2021.

Appendix A

Syntax

$$
\begin{array}{rll}
A, B, t, u, i, j \in \text{Term} ::= & * \mid \square \mid \square_{\mathbf{I}} & \text{sort} \\
& \mid \prod x : A.B \mid \lambda x.t \mid x \mid t\ u & \text{lambda-calculus} \\
& \mid \sum x : A.B \mid \langle t, u \rangle \mid t.\pi_1 \mid t.\pi_2 & \text{dependent pair} \\
& \mid \mathbf{I} \mid \mathbf{i}_0 \mid \mathbf{i}_1 \mid -i \mid i \wedge j \mid i \vee j & \text{interval} \\
& \mid t =_A u \mid \lfloor t \rfloor \mid \mathbf{Elim}_{=}(A, t, u, e, i) & \text{equality} \\
& \mid [i.A] \triangleright t & \text{coercion}
\end{array}
$$

$$\Gamma \in \text{Context} ::= \cdot \mid \Gamma, x : A$$

Appendix B

Typing Rules

B.1 Specification

$$\mathcal{S} = \{*, \Box, \Box_{\mathbf{I}}\}$$
$$\mathcal{A} = \{(*, \Box)\}$$
$$\mathcal{R} = \{(*,*,*), (*,\Box,\Box), (\Box,*,*), (\Box,\Box,\Box), (\Box_{\mathbf{I}},*,*), (\Box_{\mathbf{I}},\Box,\Box)\}$$

B.2 Standard Pure Type System Rules

$$\frac{}{\cdot \vdash \mathbf{ctx}}\ \textsc{EmptyCtx}$$

$$\frac{\Gamma \vdash \mathbf{ctx} \qquad \Gamma \vdash A : s \qquad s \in \mathcal{S} \qquad x \notin \Gamma}{\Gamma, x : A \vdash \mathbf{ctx}}\ \textsc{ExtendCtx}$$

$$\frac{\Gamma \vdash \mathbf{ctx} \qquad (s_0, s_1) \in \mathcal{A}}{\Gamma \vdash s_0 : s_1}\ \textsc{Sort}$$

$$\frac{\Gamma \vdash A : s_0 \qquad \Gamma, x : A \vdash B : s_1 \qquad (s_0, s_1, s_2) \in \mathcal{R}}{\Gamma \vdash \prod x : A.B : s_2}\ \textsc{Pi}$$

$$\frac{\Gamma, x : A \vdash t : B \qquad \Gamma \vdash \prod x : A.B : s}{\Gamma \vdash \lambda x.t : \prod x : A.B}\ \textsc{Abs}$$

$$\frac{\Gamma \vdash t : \prod x : A.B \qquad \Gamma \vdash u : A}{\Gamma \vdash t\ u : B[x := u]}\ \textsc{App}$$

$$\frac{\Gamma \vdash \mathbf{ctx} \qquad x : A \in \Gamma}{\Gamma \vdash x : A}\ \textsc{Var}$$

$$\frac{\Gamma \vdash t : A \qquad \Gamma \vdash B : s \qquad \Gamma \vdash A \equiv B : s}{\Gamma \vdash t : B}\ \textsc{Conv}$$

B.3 The Sigma Type

$$\frac{\Gamma \vdash A : * \qquad \Gamma, x : A \vdash B : *}{\Gamma \vdash \sum x : A.B : *}\ \textsc{Sigma}$$

$$\frac{\Gamma \vdash t : A \qquad \Gamma \vdash u : B[x := t]}{\Gamma \vdash \langle t, u \rangle : \sum x : A.B}\ \textsc{Pair}$$

$$\frac{\Gamma \vdash t : \sum x : A.B}{\Gamma \vdash t.\pi_1 : A}\ \textsc{Fst}$$

$$\frac{\Gamma \vdash t : \sum x : A.B}{\Gamma \vdash t.\pi_2 : B[x := t.\pi_1]}\ \textsc{Snd}$$

B.4 Cubical constructs

$$\frac{}{\Gamma \vdash \mathbf{I} : \square_{\mathbf{I}}}\ \mathrm{I} \qquad \frac{}{\Gamma \vdash \mathbf{i}_0 : \mathbf{I}}\ \mathrm{I0} \qquad \frac{}{\Gamma \vdash \mathbf{i}_1 : \mathbf{I}}\ \mathrm{I1}$$

$$\frac{\Gamma \vdash i : \mathbf{I} \qquad \Gamma \vdash j : \mathbf{I}}{\Gamma \vdash i \wedge j : \mathbf{I}}\ \textsc{Min} \qquad \frac{\Gamma \vdash i : \mathbf{I} \qquad \Gamma \vdash j : \mathbf{I}}{\Gamma \vdash i \vee j : \mathbf{I}}\ \textsc{Max}$$

$$\frac{\Gamma \vdash i : \mathbf{I}}{\Gamma \vdash -i : \mathbf{I}}\ \textsc{Neg}$$

$$\frac{\Gamma \vdash A : \mathbf{I} \to * \qquad \Gamma \vdash t : A\ \mathbf{i}_0 \qquad \Gamma \vdash u : A\ \mathbf{i}_1}{\Gamma \vdash t =_A u : *}\ \textsc{Eq}$$

$$\frac{\Gamma \vdash A : \mathbf{I} \to * \qquad \Gamma \vdash p : \prod i : \mathbf{I}.A\ i}{\Gamma \vdash \lfloor p \rfloor : (p\ \mathbf{i}_0) =_A (p\ \mathbf{i}_1)}\ \textsc{EqIntro}$$

$$\frac{\begin{array}{ll}\Gamma \vdash A : \mathbf{I} \to * & \Gamma \vdash t : A\ \mathbf{i}_0 \\ \Gamma \vdash u : A\ \mathbf{i}_1 & \Gamma \vdash e : t =_A u \qquad \Gamma \vdash i : \mathbf{I}\end{array}}{\Gamma \vdash \mathbf{Elim}_{=}(A, t, u, e, i) : A\ i}\ \textsc{EqElim}$$

$$\frac{\Gamma, i : \mathbf{I} \vdash A : s \qquad \Gamma \vdash t : A[i := \mathbf{i}_0] \qquad s \in \{*, \square\}}{\Gamma \vdash [i.A] \triangleright t : A[i := \mathbf{i}_1]}\ \textsc{Coe}$$

Appendix C

Reduction Rules

$$\frac{}{-\mathbf{i}_0 \Rightarrow \mathbf{i}_1}\ \textsc{Neg0} \qquad \frac{}{-\mathbf{i}_1 \Rightarrow \mathbf{i}_0}\ \textsc{Neg1}$$

$$\frac{}{\mathbf{i}_0 \wedge i \Rightarrow \mathbf{i}_0}\ \textsc{Min0L} \qquad \frac{}{i \wedge \mathbf{i}_0 \Rightarrow \mathbf{i}_0}\ \textsc{Min0R}$$

$$\frac{}{\mathbf{i}_1 \wedge i \Rightarrow i}\ \textsc{Min1L} \qquad \frac{}{i \wedge \mathbf{i}_1 \Rightarrow i}\ \textsc{Min1R}$$

$$\frac{}{\mathbf{i}_0 \vee i \Rightarrow i}\ \textsc{Max0L} \qquad \frac{}{i \vee \mathbf{i}_0 \Rightarrow i}\ \textsc{Max0R}$$

$$\frac{}{\mathbf{i}_1 \vee i \Rightarrow \mathbf{i}_1}\ \textsc{Max1L} \qquad \frac{}{i \vee \mathbf{i}_1 \Rightarrow \mathbf{i}_1}\ \textsc{Max1R}$$

$$\frac{}{\langle t, u\rangle.\pi_1 \Rightarrow t}\ \textsc{FstRed} \qquad \frac{}{\langle t, u\rangle.\pi_2 \Rightarrow u}\ \textsc{SndRed}$$

$$\frac{}{(\lambda x.t)u \Rightarrow t[x := u]}\ \textsc{Beta} \qquad \frac{}{[i.\mathbf{I}*] \triangleright t \Rightarrow t}\ \textsc{CoeStar}$$

$$\frac{g := \lambda j.\lambda x.[i.A[i := j \vee -i]] \triangleright x}{[i.\prod x : A.B] \triangleright f \Rightarrow \lambda x.[j.B[i := j, x := g\ j\ x]] \triangleright f\ (g\ \mathbf{i}_0\ x)}\ \textsc{CoePi}$$

$$\frac{g := \lambda j.\lambda t.[i.A[i := j \wedge i]] \triangleright t}{[i.\sum x : A.B] \triangleright t \Rightarrow \langle g\ \mathbf{i}_1\ (t.\pi_1), [j.B[i := j, x := g\ j\ (t.\pi_1)]] \triangleright t.\pi_2\rangle}\ \textsc{CoeSigma}$$

Appendix D

Conversion Rules

D.1 Reduction Equivalence

$$\frac{\Gamma \vdash t : A}{\Gamma \vdash t \equiv t : A}\ \textsc{Refl}$$

$$\frac{\Gamma \vdash t \equiv t' : A}{\Gamma \vdash t' \equiv t : A}\ \textsc{Sym}$$

$$\frac{\Gamma \vdash t \equiv t' : A \qquad \Gamma \vdash t' \equiv t'' : A}{\Gamma \vdash t \equiv t'' : A}\ \textsc{Trans}$$

$$\frac{\Gamma \vdash t \Rightarrow u}{\Gamma \vdash t \equiv u}\ \textsc{RedConv}$$

D.2 Eta Rules

$$\frac{\Gamma \vdash f : \prod x : A.B}{\Gamma \vdash f \equiv \lambda x.f\ x : \prod x : A.B}\ \textsc{PiEta}$$

$$\frac{\Gamma \vdash t : \sum x : A.B}{\Gamma \vdash t \equiv \langle t.\pi_1, t.\pi_2 \rangle : \sum x : A.B}\ \textsc{SigmaEta}$$

$$\frac{\Gamma \vdash i : \mathbf{I}}{-(-i) \equiv i : \mathbf{I}}\ \textsc{DoubleNeg}$$

$$\frac{\Gamma \vdash i : \mathbf{I} \qquad \Gamma \vdash j : \mathbf{I}}{i \vee j \equiv j \vee i : \mathbf{I}}\ \textsc{MaxComm}$$

$$\frac{\Gamma \vdash i : \mathbf{I} \qquad \Gamma \vdash j : \mathbf{I}}{i \wedge j \equiv j \wedge i : \mathbf{I}}\ \textsc{MinComm}$$

$$\frac{\Gamma \vdash e, e' : t =_A u}{\Gamma \vdash e \equiv e' : t =_A u}\ \textsc{UIP}$$

D.3 Regularity

$$\frac{\Gamma, i : \mathbf{I} \vdash A : s \qquad s \in \{*, \Box\} \qquad \Gamma \vdash t : A[i := \mathbf{i}_0] \qquad \Gamma \vdash A[i := \mathbf{i}_0] \equiv A[i := \mathbf{i}_1] : s}{\Gamma \vdash [i.A] \triangleright t \equiv t : A[i := \mathbf{i}_1]}\ \textsc{CoeId}$$

D.4 Congruence Rules

$$\frac{\Gamma \vdash A \equiv A' : s_1 \qquad \Gamma \vdash B \equiv B' : s_2}{\Gamma \vdash \prod x : A.B \equiv \prod x : A'.B' : s_2}\ \textsc{PiCong}$$

$$\frac{\Gamma, x : A \vdash t \equiv t' : B}{\Gamma \vdash \lambda x.t \equiv \lambda x.t' : \prod x : A.B}\ \textsc{LamCong}$$

$$\frac{\Gamma \vdash f \equiv f' : \prod x : A.B \qquad \Gamma \vdash t \equiv t' : A}{\Gamma \vdash f\ t \equiv f'\ t' : B[x := t]}\ \textsc{AppCong}$$

$$\frac{\Gamma \vdash A \equiv B' : * \qquad \Gamma \vdash A \equiv B' : *}{\Gamma \vdash \sum x : A.B \equiv \sum x : A'.B' : *}\ \textsc{SigmaCong}$$

$$\frac{\Gamma \vdash t \equiv t' : A \qquad \Gamma \vdash u \equiv u' : B[x := t]}{\Gamma \vdash \langle t, u \rangle \equiv \langle t', u' \rangle : \sum x : A.B}\ \textsc{PairCong}$$

$$\frac{\Gamma \vdash t \equiv u : \sum x : A.B}{\Gamma \vdash t.\pi_1 \equiv u.\pi_1 : A}\ \textsc{FstCong}$$

$$\frac{\Gamma \vdash t \equiv u : \sum x : A.B}{\Gamma \vdash t.\pi_2 \equiv u.\pi_2 : B[x := t.\pi_1]}\ \textsc{SndCong}$$

$$\frac{\Gamma \vdash i \equiv i' : \mathbf{I}}{\Gamma \vdash -i \equiv -i' : \mathbf{I}}\ \textsc{NegCong}$$

$$\frac{\Gamma \vdash i \equiv i' : \mathbf{I} \qquad \Gamma \vdash j \equiv j' : \mathbf{I}}{\Gamma \vdash i \wedge j \equiv i' \wedge j' : \mathbf{I}}\ \textsc{MinCong}$$

$$\frac{\Gamma \vdash i \equiv i' : \mathbf{I} \qquad \Gamma \vdash j \equiv j' : \mathbf{I}}{\Gamma \vdash i \vee j \equiv i' \vee j' : \mathbf{I}}\ \textsc{MaxCong}$$

$$\frac{\Gamma, i : \mathbf{I} \vdash A \equiv A' : s \qquad \Gamma \vdash t \equiv t' : A[i := \mathbf{i}_0] \qquad s \in \{*, \Box\}}{\Gamma \vdash [i.A] \triangleright t \equiv [i.A'] \triangleright t' : A[i := \mathbf{i}_1]}\ \textsc{CoeCong}$$

$$\frac{\Gamma \vdash A \equiv A' : \mathbf{I} \to * \qquad \Gamma \vdash t \equiv t' : A\ \mathbf{i}_0 \qquad \Gamma \vdash u \equiv u' : A\ \mathbf{i}_1}{\Gamma \vdash t =_A u \equiv t' =_{A'} u' : *}\ \textsc{EqCong}$$

$$\frac{\Gamma \vdash A \equiv A' : \mathbf{I} \to * \qquad \Gamma \vdash p \equiv p' : \prod i : \mathbf{I}.A\ i}{\Gamma \vdash \lfloor p \rfloor \equiv \lfloor p' \rfloor : p\ \mathbf{i}_0 =_A p\ \mathbf{i}_1}\ \textsc{EqIntroCong}$$

$$\frac{\begin{array}{cc}\Gamma \vdash A \equiv A' : \mathbf{I} \to * & \Gamma \vdash u \equiv u' : A\ \mathbf{i}_1 \\ \Gamma \vdash t \equiv t' : A\ \mathbf{i}_0 & \Gamma \vdash i \equiv i' : \mathbf{I}\end{array}}{\Gamma \vdash \mathbf{Elim}_=(A, t, u, e, i) \equiv \mathbf{Elim}_=(A', t', u', e', i') : A\ i}\ \textsc{EqElimCong}$$