CHALMERS
UNIVERSITY OF TECHNOLOGY

UNIVERSITY OF GOTHENBURG

A Logical Relation for Dependent Type Theory Formalized in Agda

Master's thesis in Computer Science

JOAKIM ÖHMAN

Department of Computer Science and Engineering
CHALMERS UNIVERSITY OF TECHNOLOGY
UNIVERSITY OF GOTHENBURG
Gothenburg, Sweden 2016

Master's thesis 2016

A Logical Relation for Dependent Type Theory Formalized in Agda

With Application to Canonicity and Pi-Injectivity

JOAKIM ÖHMAN

Department of Computer Science and Engineering
Division of Computer Science
Programming Logic Group
Chalmers University of Technology
University of Gothenburg
Gothenburg, Sweden 2016

A Logical Relation for Dependent Type Theory Formalized in Agda
With Application to Canonicity and Pi-Injectivity
JOAKIM ÖHMAN

Supervisor: Andrea Vezzosi, Department of Computer Science and Engineering
Cosupervisor: Andreas Abel, Department of Computer Science and Engineering
Examiner: Patrik Jansson, Department of Computer Science and Engineering

Master's Thesis 2016
Department of Computer Science and Engineering
Division of Computer Science
Programming Logic Group

Chalmers University of Technology
SE-412 96 Gothenburg
Telephone +46 31 772 1000

University of Gothenburg
SE-405 30 Gothenburg
Telephone +46 31 786 0000

Typeset in LaTeX
Gothenburg, Sweden 2016

A Logical Relation for Dependent Type Theory Formalized in Agda
With Application to Canonicity and Pi-Injectivity
JOAKIM ÖHMAN
Department of Computer Science and Engineering
Chalmers University of Technology
University of Gothenburg

Abstract

When writing proofs, it is desirable to show that one's proof is correct. With formalising a proof in dependent type theory, it is implied that the proof is correct as long as the type theory is correct.

This thesis covers the formalisation of a fragment of Martin-Löf type theory with one universe, for which its grammar and judgements are formalised. A logical relation is then defined as well as validity judgements and proven valid with respect to the type system. This is then used to prove canonicity and injectivity of the Π-type.

Keywords: type theory, dependent types, logical relation, Agda, formalization.

Acknowledgements

I would like to thank my supervisors, Andreas Abel and Andrea Vezzosi, for their assistance of helping with definitions and proofs, their support and expertise in the area.

Joakim Öhman, Gothenburg, November 2016

Contents

List of Figures and Tables

1 Introduction

When writing proofs, it is often desirable to ensure that ones proof is correct. Usually, a proof has to be peer reviewed before it can be deemed correct, however mistakes can still remain undiscovered. One way to verify the correctness of a proof is to formalise it in a formal language which has a computer program to verify the proof.

Some type theories functions well as formal languages to formalise proofs. In this kind of theory we have terms of types instead of elements of sets. The difference is that with these types we can represent logical concepts, for example function types can represent an implication. The types of type theory are also similar to the types of programming.

In Martin-Löf type theory we can represent predicate logic with the dependent types, which are types that contain values. With the dependent product type Π we can represent universal quantification, and with the dependent sum type Σ we can represent existential quantification. This makes this type system ideal for proving various properties of systems. There are also proof assistant languages such as Coq and Agda, which are based on this type theory, where we can verify proofs by computer.

This thesis aim to formalise a logical relation of a simplified Martin-Löf type theory language in Agda. There is also other works on the topic of type theory in type theory, such as Chapman [1] [2] and Altenkirch/Kaposi [3].

1.1 Goal

In this thesis the goal is to prove and formalize in Agda a logical relation for a simple Martin-Löf type theory language, with a dependent product type, a single universe of types a la Russel and natural numbers with natural recursion. This logical relation can then be used to prove certain properties, such as Π injectivity and canonicity. The approach for the proof will be mostly based on Abel/Scherer [4] and the abstract by Abel/Coquand/Mannaa [5].

1.2 Thesis outline

In this introduction chapter, a general introduction to the topic of formalisation and type theory is given as well as the goal of the thesis. In the second chapter, we introduce more of the concept of formalisation, describe Martin-Löf type theory and Agda. The proof and formalisation methods are also described.

In the third chapter, the formalisation of the grammar and judgements are presented as well as their lemmas. In the fourth chapter, the logical relation and its lemmas are introduced. Also, the validity judgements are presented with its lemmas and finally the fundamental theorem is shown with its consequences.

In the fifth chapter, the formalisation is discussed along with possible future applications and extensions. In the last and sixth chapter are the conclusions of the thesis.

2

Background

This chapter will introduce and describe various prerequisites to understand this thesis. First, the concept of formalisation is introduces with some formal languages relevant to the thesis. Second, concepts of equality in type theory are introduced. Third, the language used to formalise the proofs is introduced. Last, the proof methods of this thesis are described.

2.1 Formalisation

Most often, proofs are described in a natural language. While natural languages are the most natural way to convey ideas, it is infeasible to verify such a proof by machine. As such, it is required that someone proof-reads the proofs to ensure its soundness, however mistakes can be missed, especially if the proof is complex. This is where the idea of formalisation comes in.

Formalisation is the act of writing a proof in a formal language. These languages have procedures to verify the an instance of it, some of the procedures being verification by grammar and by rules. This can give us confidence that the proof is correct.

There are also computer programs that can verify an instance of a formal language. Programming languages are themselves formal languages, which are usually verified by its interpreter, compiler or type checking program. For proofs, these programs are referred to as proof assistants, which usually also assists with the writing of a proof.

Formal proofs are however not guaranteed to be error free. While the proofs themselves may be verified, the definitions may still contain errors. The definition can be too strong and allow anything to be proven when it is assumed, it could be too weak and make it always provable, or anything in between that the author did not have in mind. Hence, it is still important to be precise with the definitions.

2.2 Lambda calculus

Lambda calculus is one of these formal languages, which is usually used to describe computations. Lambda calculus was first introduced by Alonzo Church in 1936.

2.2.1 Untyped lambda calculus

The language of untyped lambda calculus is a variant of lambda calculus without a notion of typing [6, chapter 5, p. 51]. It has the following grammar of lambda terms:

$$t, u ::= x \mid \lambda x.t \mid t\,u$$

where x are variables, $\lambda x.t$ are lambda abstractions and $t\,u$ are term applications. Lambda abstractions can be seen as anonymous functions, which takes one argument. Applications are the application of an argument to one of these anonymous functions.

Variables can either be free or bound. A variable is bound if it is introduces as a function variable by a lambda abstraction. Any other variable is known as free. For instance, in $\lambda x.(\lambda y.x\,y\,z)$ the variable z is free while x and y are bound.

The computation of lambda terms can be described by the following *reduction rules*:

$$\frac{t \Rightarrow t'}{t\,u \Rightarrow t'\,u} \qquad\qquad \frac{}{(\lambda x.t)\,u \Rightarrow t[u/x]}$$

where $t[u/x]$ is the substitution where each occurrence of x in t is substituted with u.

These rules tell us how a lambda term can be reduced, usually in multiple steps, towards a normal form. Those are terms which cannot be reduced any further. Here, these are free variables, lambda abstractions and applications with the function being in normal form.

We presented here a weak head reduction, which has some subtle differences with full reduction. For a fully reduced term, every part of a lambda term is fully reduced, while for weak head reduction, reduction stops when the head can be no longer reduced.

However, not all lambda terms in untyped lambda calculus reduce to a normal form, and as such never terminate. This is something that is not desirable for a proving system, as without guaranteed termination we can say that we arrive at a proof in infinite steps, hence anything would be provable. However, that this fact holds for untyped lambda calculus along with other things makes it Turing complete.

2.2.2 Simply typed lambda calculus

Simply typed lambda calculus (STLC) is an extension of untyped lambda calculus with the concept of types [6, chapter 9, p. 99]. This means that every term needs to have a type, by satisfying the typing rules, to be valid. This also includes free variable, which means that they needs to be valid in a context.

The grammar of types are as follows:

$$A, B ::= X \mid A \to B$$

where X is a type of a variable in the related context. The context has the following grammar:

$$\Gamma ::= \epsilon \mid \Gamma, x : A$$

where ϵ is the empty context and $\Gamma, x : A$ extends context Γ with a variable x of type A.

STLC has the following typing rules:

$$\frac{\Gamma \vdash t : A \to B \qquad \Gamma \vdash u : A}{\Gamma \vdash t\, u : B} \qquad\qquad \frac{\Gamma, x : A \vdash t : B}{\Gamma \vdash \lambda(x : A).t : A \to B}$$

Note that in STLC the lambda abstractions have there variables type annotated.

A consequence of the typing rules is that terms that do not reduce to normal form are not well typed, and as such all valid terms terminate. As a consequence this system is not Turing complete. This makes this system more suitable for proofs, however the typing rules are very limited for this.

2.3 Martin-Löf Type Theory

One of the formal languages one can write proofs in is Martin-Löf type theory (MLTT). MLTT is a type theory, an alternative approach to the foundation mathematics, where the notion of types are used as logical constructs and sets. The logical constructs are here part of the language, unlike classical mathematics where set theory is disjoint with predicate logic. [7]

MLTT is built on top of STLC, inheriting most of its constructs. However, in MLTT, both types and terms live in the same grammar. This means that types can depend on terms and types can be denoted as a term of a type. As such, it is necessary to have a type of types, also called universe.

The main feature introduced by MLTT is dependent types. These types are introduced by taking two types A and B, where B can depend on an instance of A. There are two kinds of these:

- Π-types, also known as dependent product types, are effectively function types of STLC with dependency functionality; given types A and B then $\Pi\,(a : A)\, B[a]$. This says that for all instances of a of A, if $B[a]$ holds for all these a, then $\Pi\,(a : A)\, B[a]$ holds.
- Σ-types, also know dependent sum types. Similar to Π types, this takes two types A and B such that $\Sigma(a : A)B[a]$ is a *Sigma*-type. This says that if there exists an a of A such that $B[a]$ holds, then $\Sigma(a : A)B[a]$ holds.

Note that above we used $B[a]$. This is the same as a substitution in untyped lambda calculus, only that here we have left out the specific variable to substitute with.

2.4 Concepts of equality

In this thesis we will work with a number of different notions of equality. Here we will present and describe them.

2.4.1 Propositional equality

For MLTT, if one were to introduce natural number addition, it would not immediately follow that commutativity would hold, that is $a + b$ is equal to $b + a$, however if one were to substitute them with number, it would hold. This is since the two terms reduced are equal, yet it is not immediate from arbitrary numbers.

With identity types, it is possible to introduce a proof of the above. If we let $\mathsf{Id_A}$ be an identity type for type A, then $\Pi\,(a : A)\,\mathsf{Id_A}\,a\,a$ is the base case of reflexivity. From this base case, the properties of symmetry (if $\mathsf{Id_A}\,a\,b$ then $\mathsf{Id_A}\,b\,a$), transitivity (if $\mathsf{Id_A}\,a\,b$ and $\mathsf{Id_A}\,b\,c$ then $\mathsf{Id_A}\,a\,c$) and congruence (if $\mathsf{Id_A}\,a\,b$ then $\mathsf{Id_A}\,(f\,a)\,(f\,b)$) can be derived. By this and recursion, it is possible to prove the commutativity property of addition.

It is the identity types that create these propositional equalities. Note however that the terms of the equality needs to be of the same type, which can create complexities if they are of different types that are equal.

2.4.2 Heterogeneous equality

As mentioned in the propositional equality section, that equality cannot prove the equality of term if they have different types. This is where heterogeneous equality comes in.

Heterogeneous equality is an extension of propositional equality such that if the types of the terms are propositionally equal, then the terms can be heterogeneously equal. It is also possible to take a heterogeneous equality and transform it into a propositional equality.

2.4.3 Conversion

Conversion is a defined equality in a system, such that convertible elements can be substituted with each other while the relevant properties are still the same. Which properties that are deemed relevant depends on the system. However, the equality properties of reflexivity, symmetry and transitivity must hold, as otherwise it would not be a conversion relation.

In this thesis, conversion rules are defined for the language for which proofs are formalised for.

2.5 Agda

In this thesis the proofs are formalised in Agda, which is a proof assistant and a programming language.[8] How it functions as both is that any type works as a proposition and any function of that type works as a proof of the proposition, as by the Curry-Howard isomorphism.

Agda is also referred to as a dependently typed functional language. The dependent types are inherited from MLTT.

The Agda language lets one define a function with a type and an instance of that type. This function is then given a name, so that it can be called recursively. However, this form of recursion lets one introduce non-terminating functions. To solve this, Agda uses a termination checker.

The termination checker checks that recursive calls made are reduced. This means that if we have a function f which accepts natural numbers, then $f\,n$ is a smaller call than $f\,(n+1)$. As such, termination is ensured.

Agda also allows mutually dependent definitions. This allows us to define relation a large amount of cross dependencies. It also allows us to prove properties that are mutually dependent.

2.6 Methods

In this section, the different proof methods are described.

2.6.1 Logical relations

When making proofs for complex predicates, it is often desirable to show that certain properties holds for certain parts of the predicate. As such, one can introduce a new predicate which holds these desired properties in that certain part. This new predicate is what we refer to as a logical relation.

Logical relations are inductively defined on the type of the predicate. This means that certain properties can be linked to certain terms, strengthening the typing relation.

The notion of a logical relation does not have precise definition, and as such may differ between works. One early work using logical relations is by Plotkin [9].

A proof by logical relation involves showing that one can from an original predicate show that the logical relation holds and vice versa. By proving this relation, one can extract the properties of logical relation and show that they hold from the original predicate.

2.6.2 De Bruijn index

When formalising proofs with contexts, it is desirable to use a more simple variable representation than names. This is basically what De Bruijn indices lets us do; it uses natural numbers to represent variables along with changes to variable binders [6, chapter 6, p. 75].

Consider the following typed context:

$$\Gamma, c : C, b : B, a : A$$

With De Bruijn indices, each name in the above context is replaced with a natural number, starting with zero from the right. As such the above become this when translated:

$$\Gamma, 2 : C, 1 : B, 0 : A$$

Usually the numbers are omitted as they can be implicitly derived.

For any construct that introduces a variable in scope, with De Bruijn indices, the variable is removed, such that for λ-abstractions, they are denoted as $\lambda\, t$ for term t. These function such that the binder introduces a new variable as zero and shift the rest by one. As such, for the followings:

$$\Gamma, 2 : C, 1 : B, 0 : A \vdash \lambda\, t : F \to G$$

variables in t are in the order as follows:

$$\Gamma, 3 : C, 2 : B, 1 : A, 0 : F \vdash t : G$$

This can also nested, which can make it complicated to follow which variable corresponds to which binder, however this simplified the implementation of a proof formulation.

2.6.3 Induction-recursion

Sometimes in type theory, it is necessary to define a type and a proposition using that type at the same time. This means that the type and the proposition are mutually dependent. This is a feature of Agda and was first investigated by Dybjer [10].

In Agda, when introducing a data type, it must only contain strictly-positive inductive instances. This means that the type may not have an instance where one of the premises is a function which domain is the inductive type. If this were the case, it would be possible to introduce a proof which does not terminate.

To be able to define certain relations which requires a sense of negative occurence, the induction-recursion method can be used. This method makes a part of a relation inductive, with the rest recurse over it, and matching cases to the inductive relation to form the desired relation.

3 Language

In this chapter we present the grammar of the language as well as its judgements. We will introduce functions over the grammar and some of their properties. The judgements of the language are *being a type*, *term being of type*, type and term conversion, and type and term reduction. Additionally, we will also prove some lemmas of these judgements.[1]

3.1 Grammar

Below we present the grammar of our language.[2]

$$A, B, t, u, n ::= \mathsf{U} \mid \mathbb{N} \mid \Pi\, A\, B \mid x \mid \lambda\, t \mid t\, u \mid \mathsf{zero} \mid \mathsf{suc}\, n \mid \mathsf{natrec}\, A\, t\, u\, n$$

$$x ::= \mathsf{x_0} \mid \mathsf{x_1} \mid \mathsf{x_2} ...$$

Here we have the lambda-calculus' constructs variable x, lambda abstraction $\lambda\ t$ and application $t\ u$. $\Pi\ A\ B$ is the dependent product type and U is the universe of types. $\mathbb{N}$ is the type of natural numbers with zero and the successor construct suc. To be able to do more interesting computations, we also have natural recursion as $\mathsf{natrec}\, G\, s\, z\, n$ where G is the result type function, s is the successor case, z the zero case and n is the natural number to recurse over.

De Bruijn indices will be used for the grammar. As such, variables are represented with natural numbers. To make sure they are not confused with numbers as part of a grammar, $\mathsf{x_n}$ is used instead, where n is the index of the variable. The binders in this grammar are t in $\lambda\, t$, B in $\Pi\, A\, B$ and G in $\mathsf{natrec}\, G\, s\, z\, n$.

3.1.1 Weakening

Similarly to the transition between untyped and simply typed lambda calculus, the judgements presented later will require a context of variable types. As such, it is

[1] The formalization is available online at `https://mr-ohman.github.io/logrel-mltt/` with the complete definitions and proofs.

[2] Available in the code as `Term` in module `Definition.Untyped`

desirable to show that for a valid term of a context, the term is also valid for an extended context. To prepare for this, we introduce the concept of weakening of terms.

Weakening essentially shifts the variable indices to accommodate for a new context. With this, properties of minimal context can be shown to hold true also in for larger contexts.[3]

We also introduce a composition operator $\rho \cdot \rho'$ so that two weakenings can be composed. The composition of certain weakening constructors is presented with the constructors.

We have three constructors of weakening operations:

- id, the identity weakening. This has the property such that for any term t, $\mathsf{id}\, t$ is propositionally the same as t. It is indeed the identity with regard to composition, as when a weakening ρ is composed with id in any order, it is propositionally equal to ρ.
- $\mathsf{step}\, \rho$, which when applied to a term shifts all variables by one and then applies the weakening ρ. It essentially adds one more element to a context. For composition, it has the property that $\mathsf{step}\, \rho \cdot \rho'$ is equal to $\mathsf{step}\, (\rho \cdot \rho')$.
- $\mathsf{lift}\, \rho$, which "lifts" the the top-most variable and then applies ρ for the rest of the variables of the term. This has the composition properties:
  - $\mathsf{lift}\, \rho \cdot \mathsf{lift}\, \rho'$ is propositionally equal to $\mathsf{lift}\, (\rho \cdot \rho')$
  - $\mathsf{lift}\, \rho \cdot \mathsf{step}\, \rho'$ is equal to $\mathsf{step}\, (\rho \cdot \rho')$.
  - $\mathsf{lift}^n\, \mathsf{id}$ is equal to id, where n is an arbitrary number of lift applications.

We also introduce the shorthand $\mathsf{wk1}$, which is a synonym of $\mathsf{step}\, \mathsf{id}$. Additionally, with weakening, the non-dependent function type $A \to B$ can be introduced, which is a synonym to $\Pi\, A\, (\mathsf{wk1}\ B)$.

3.1.2 Substitution

Substitution is simply the act of substituting variables in a term with other terms.

We implement parallel substitution, which essentially means that all variables are substituted in one application. The representation of substitutions we use are functions from natural numbers to terms. Each free variable is then substituted using this function.[4]

Substitutions can, like terms, be weakened. We also introduce the operation purge, which applies a weakening to the indices of a substitution.

We define the substitution function $t[u]$, which is a special case of parallel substitution, which substitutes x_0 in t with u and shifts the rest of the variable indices

[3] Weakening is represented in the code as `Wk` in module `Definition.Untyped`

[4] Substitution is represented in the code as `Subst` in module `Definition.Untyped`

downwards by one. This is mostly used when we have a binder that has a term applied, for instance with $(\lambda\ t)\ u$ it will reduce to $t[u]$.

We also define a variant $t[u]{\uparrow}$, which as $t[u]$ substitutes x_0 in t with u, however it does not shift the other variable indices. This is used when we want to apply something to x_0 with it still being the top-most variable, such as when we want to apply a function to a variable.

3.1.3 Properties

As weakening and substitution are functions over the syntax and not a member of the syntax, we do not need to provide equality rules for these. Instead, we need to provide equality proofs using propositional equality. We will for instance need to prove that a weakening is equal to its substitution counterpart when applied to a term.[5]

Lemma 3.1.1. *$(\rho \cdot \rho')\ t$ is propositionally equal to $\rho\ (\rho'\ t)$.*

Proof. By recursion over t using congruence. In the variable case by recursion over the weakenings. □

Lemma 3.1.2. *Given weakening ρ it follows that $\mathsf{wk1}\ \cdot \rho$ is propositionally equal to $\mathsf{lift}\ \rho\ \cdot \mathsf{wk1}$.*

Proof. By recursion over ρ. □

Lemma 3.1.3. *Given weakening ρ and substitution σ it holds that $\rho\ (\sigma\ t)$ is propositionally equal to $(\rho\ \sigma)\ t$.*

Proof. By recursion over t. □

Lemma 3.1.4. *For expression t, weakening ρ and substitution σ it follows that $\sigma\ (\rho\ t)$ is propositionally equal to $(\mathsf{purge}\ \rho\ \sigma)\ t$.*

Proof. By recursion over t. □

Lemma 3.1.5. *$\rho\ (t[a])$ is propositionally equal to $((\mathsf{lift}\ \rho)\ t)[\rho\ a]$ and $\rho\ (t[a]{\uparrow})$ propositionally equal to $((\mathsf{lift}\ \rho)\ t)[(\mathsf{lift}\ \rho)\ a]{\uparrow}$.*

Proof. By lemma 3.1.3 and 3.1.4 with recursion over t. □

3.2 Judgements

In this section we present the judgements of our language. Each judgement takes a context of terms as its first parameter, which is essentially a list of types with ϵ

[5]In the code, these properties are proved in module `Definition.Untyped.Properties`

being the empty context and Γ, A being a context Γ extended by term A.[6]

The different judgements tell us which constructs are well-formed. These are $\vdash \Gamma$ for contexts, $\Gamma \vdash A$ for types, $\Gamma \vdash t : A$ for terms, $\Gamma \vdash A = B$ and $\Gamma \vdash t = u : A$ for type and term conversion respectively. There is also the rules for the well-formed variables $\mathsf{x_n} : A \in \Gamma$.[7]

Certain premises are highlighted below in the rules. These are additional premises, necessary to be able to prove certain lemmas.

Well-formed contexts:

$$\frac{}{\vdash \epsilon} \qquad\qquad \frac{\vdash \Gamma \qquad \Gamma \vdash A}{\vdash \Gamma, A}$$

Variables:

$$\frac{}{\mathsf{x_0} : \mathsf{wk1}\ A \in \Gamma, A} \qquad\qquad \frac{\mathsf{x_n} : A \in \Gamma}{\mathsf{x_{n+1}} : \mathsf{wk1}\ A \in \Gamma, B}$$

Types:

$$\frac{\vdash \Gamma}{\Gamma \vdash \mathbb{N}}\,\mathbb{N} \qquad \frac{\vdash \Gamma}{\Gamma \vdash \mathsf{U}}\,\mathsf{U} \qquad \frac{\Gamma \vdash F \qquad \Gamma, F \vdash G}{\Gamma \vdash \Pi\, F\, G}\,\Pi \qquad \frac{\Gamma \vdash A : \mathsf{U}}{\Gamma \vdash A}\,\text{univ}$$

Terms:

$$\frac{\vdash \Gamma \qquad x : A \in \Gamma}{\Gamma \vdash x : A}\,\text{var} \qquad\qquad \frac{\Gamma \vdash F : \mathsf{U} \qquad \Gamma, F \vdash G : \mathsf{U}}{\Gamma \vdash \Pi\, F\, G : \mathsf{U}}\,\Pi$$

$$\frac{\Gamma \vdash F \qquad \Gamma, F \vdash t : G}{\Gamma \vdash \lambda\, t : \Pi\, F\, G}\,\lambda \qquad\qquad \frac{\Gamma \vdash a : F \qquad \Gamma \vdash g : \Pi\, F\, G}{\Gamma \vdash g\, a : G[a]}\,\text{app}$$

$$\frac{\vdash \Gamma}{\Gamma \vdash \mathbb{N} : \mathsf{U}}\,\mathbb{N} \qquad \frac{\vdash \Gamma}{\Gamma \vdash \mathsf{zero} : \mathbb{N}}\,\text{zero} \qquad \frac{\Gamma \vdash n : \mathbb{N}}{\Gamma \vdash \mathsf{suc}\, n : \mathbb{N}}\,\text{suc}$$

$$\frac{\Gamma, \mathbb{N} \vdash G \qquad \Gamma \vdash z : F[\mathsf{zero}] \qquad \Gamma \vdash s : \Pi\, \mathbb{N}\, (G \to G[\mathsf{suc}\, \mathsf{x_0}]\uparrow) \qquad \Gamma \vdash n : \mathbb{N}}{\Gamma \vdash \mathsf{natrec}\, G\, z\, s\, n : G[n]}\,\text{natrec}$$

$$\frac{\Gamma \vdash t : A \qquad \Gamma \vdash A = B}{\Gamma \vdash t : B}\,\text{conv}$$

Of note here is the natrec rule, specifically the s premise and its type. The type $\Pi\, \mathbb{N}\, (G \to G[\mathsf{suc}\, \mathsf{x_0}]\uparrow)$ describes a function that accepts argument two arguments m and g of types $\mathbb{N}$ and $G[n]$ respectively, such that it returns a term of type $G[\mathsf{suc}\, n]$.

[6]The context type is called `Con` in the code and is defined in module `Definition.Untyped`

[7]In the code, the judgements are defined in the module `Definition.Typed`

Below continues the judgements with conversion rules.

Type conversion:

$$\frac{\Gamma \vdash A}{\Gamma \vdash A = A}\,\text{refl} \qquad \frac{\Gamma \vdash A = B}{\Gamma \vdash B = A}\,\text{sym} \qquad \frac{\Gamma \vdash A = B \qquad \Gamma \vdash B = C}{\Gamma \vdash A = C}\,\text{trans}$$

$$\frac{\Gamma \vdash A = B : \mathsf{U}}{\Gamma \vdash A = B}\,\text{univ} \qquad \frac{\Gamma \vdash F \qquad \Gamma \vdash F = H \qquad \Gamma, F \vdash G = E}{\Gamma \vdash \Pi\, F\, G = \Pi\, H\, E}\,\Pi\text{-cong}$$

Term conversion:

$$\frac{\Gamma \vdash t : A}{\Gamma \vdash t = t : A}\,\text{refl} \qquad \frac{\Gamma \vdash t = u : A}{\Gamma \vdash u = t : A}\,\text{sym} \qquad \frac{\Gamma \vdash t = u : A \qquad \Gamma \vdash u = r : A}{\Gamma \vdash t = r : A}\,\text{trans}$$

$$\frac{\Gamma \vdash t = u : A \qquad \Gamma \vdash A = B}{\Gamma \vdash t = u : B}\,\text{conv} \qquad \frac{\Gamma \vdash m = n : \mathbb{N}}{\Gamma \vdash \mathsf{suc}\, m = \mathsf{suc}\, n : \mathbb{N}}\,\text{suc-cong}$$

$$\frac{\Gamma \vdash F \qquad \Gamma \vdash F = H : \mathsf{U} \qquad \Gamma, F \vdash G = E : \mathsf{U}}{\Gamma \vdash \Pi\, F\, G = \Pi\, H\, E : \mathsf{U}}\,\Pi\text{-cong}$$

$$\frac{\Gamma \vdash f = g : \Pi\, F\, G \qquad \Gamma \vdash a = b : F}{\Gamma \vdash f\, a = g\, a : G[a]}\,\text{app-cong}$$

$$\frac{\Gamma \vdash F \qquad \Gamma, F \vdash b : G \qquad \Gamma \vdash a : F}{\Gamma \vdash (\lambda\, b)\, a = b[a] : G[a]}\,\beta\text{-red}$$

$$\frac{\Gamma \vdash F \quad \Gamma \vdash f : \Pi\, F\, G \quad \Gamma \vdash g : \Pi\, F\, G \quad \Gamma, F \vdash (\mathsf{wk1}\, f)\, \mathsf{x}_0 = (\mathsf{wk1}\, g)\, \mathsf{x}_0 : G}{\Gamma \vdash f = g : \Pi\, F\, G}\,\text{fun-ext}$$

$$\frac{\Gamma, \mathbb{N} \vdash F = F' \quad \Gamma \vdash z = z' : F[\mathsf{zero}] \quad \begin{array}{c}\Gamma \vdash s = s' :\\ \Pi\, \mathbb{N}\, (F \to F[\mathsf{suc}\, \mathsf{x}_0]\!\uparrow)\end{array} \quad \Gamma \vdash n = n' : \mathbb{N}}{\Gamma \vdash \mathsf{natrec}\, F\, z\, s\, n = \mathsf{natrec}\, F'\, z'\, s'\, n' : F[n]}$$

$$\frac{\Gamma, \mathbb{N} \vdash F \quad \Gamma \vdash z : F[\mathsf{zero}] \quad \Gamma \vdash s : \Pi\, \mathbb{N}\, (F \to F[\mathsf{suc}\, \mathsf{x}_0]\!\uparrow)}{\Gamma \vdash \mathsf{natrec}\, F\, z\, s\, \mathsf{zero} = z : F[\mathsf{zero}]}\,\text{natrec-zero}$$

$$\frac{\Gamma, \mathbb{N} \vdash F \quad \Gamma \vdash z : F[\mathsf{zero}] \quad \Gamma \vdash s : \Pi\, \mathbb{N}\, (F \to F[\mathsf{suc}\, \mathsf{x}_0]\!\uparrow) \quad \Gamma \vdash n : \mathbb{N}}{\Gamma \vdash \mathsf{natrec}\, F\, z\, s\, (\mathsf{suc}\, n) = (s\, n)\, (\mathsf{natrec}\, F\, z\, s\, n) : F[\mathsf{suc}\, n]}\,\text{natrec-suc}$$

Additionally, we have the judgements for the weak head reductions, where $\Gamma \vdash A \Rightarrow B$ and $\Gamma \vdash t \Rightarrow u : A$ are for type and term reduction respectively, and $\Gamma \vdash A \Rightarrow^* B$ and $\Gamma \vdash t \Rightarrow^* u : A$ are for type and term reduction closures.

Reductions:

$$\frac{\Gamma \vdash A \Rightarrow B : \mathsf{U}}{\Gamma \vdash A \Rightarrow B}\,\text{univ} \qquad \frac{\Gamma \vdash t \Rightarrow u : A \qquad \Gamma \vdash A = B}{\Gamma \vdash t \Rightarrow u : B}\,\text{conv}$$

$$\frac{\Gamma \vdash f \Rightarrow g : \Pi\, F\, G \qquad \Gamma \vdash a : F}{\Gamma \vdash f\, a \Rightarrow g\, a : G[a]}\,\text{app-cong}$$

$$\frac{\Gamma \vdash F \qquad \Gamma, F \vdash b : G \qquad \Gamma \vdash a : F}{\Gamma \vdash (\lambda\, b)\, a \Rightarrow b[a] : G[a]}\,\beta\text{-red}$$

$$\frac{\Gamma, \mathbb{N} \vdash F \quad \Gamma \vdash z : F[\mathsf{zero}] \quad \Gamma \vdash s : \Pi\, \mathbb{N}\, (F \to F[\mathsf{suc}\, \mathsf{x}_0]\!\uparrow) \quad \Gamma \vdash n \Rightarrow n' : \mathbb{N}}{\Gamma \vdash \mathsf{natrec}\, F\, z\, s\, n \Rightarrow \mathsf{natrec}\, F\, z\, s\, n' : F[n]}$$

$$\frac{\Gamma, \mathbb{N} \vdash F \qquad \Gamma \vdash z : F[\mathsf{zero}] \qquad \Gamma \vdash s : \Pi\, \mathbb{N}\, (F \to F[\mathsf{suc}\, \mathsf{x}_0]\!\uparrow)}{\Gamma \vdash \mathsf{natrec}\, F\, z\, s\, \mathsf{zero} \Rightarrow z : F[\mathsf{zero}]}\,\text{natrec-zero}$$

$$\frac{\Gamma, \mathbb{N} \vdash F \qquad \Gamma \vdash z : F[\mathsf{zero}] \qquad \Gamma \vdash s : \Pi\, \mathbb{N}\, (F \to F[\mathsf{suc}\, \mathsf{x}_0]\!\uparrow) \qquad \Gamma \vdash n : \mathbb{N}}{\Gamma \vdash \mathsf{natrec}\, F\, z\, s\, (\mathsf{suc}\, n) = (s\, n)\, (\mathsf{natrec}\, F\, z\, s\, n) : F[\mathsf{suc}\, n]}$$

Reduction closure:

$$\frac{\Gamma \vdash A}{\Gamma \vdash A \Rightarrow^* A}\,\text{id} \qquad \frac{\Gamma \vdash A \Rightarrow A' \qquad \Gamma \vdash A' \Rightarrow^* B}{\Gamma \vdash A \Rightarrow^* B}\,\text{step}$$

$$\frac{\Gamma \vdash t : A}{\Gamma \vdash t \Rightarrow^* t : A}\,\text{id} \qquad \frac{\Gamma \vdash t \Rightarrow t' : A \qquad \Gamma \vdash t' \Rightarrow^* u : A}{\Gamma \vdash t \Rightarrow^* u : A}\,\text{step}$$

For cases where it necessary to not only have well-formed equalities, but also the types and terms of that equality being well-formed, we introduce the notations $\Gamma \vdash A :=: B$ and $\Gamma \vdash t :=: u : A$. We refer to these as *fully well-formed* equalities.

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

Similarly to equality, we also introduce the notion of fully well-formed reduction closures, formally denoted as $\Gamma \vdash A :\Rightarrow^*: B$ and $\Gamma \vdash t :\Rightarrow^*: u : A$.

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

For the reductions, there are terms that cannot be reduced. These are called to be in weak head normal form (WHNF). A view of these are introduced. The terms that are in WHNF are U, $\mathbb{N}$, $\Pi\, A\, B$ for any A and B, $\lambda\, t$ for any t, zero, $\mathsf{suc}\, t$ for any t, and neutral term.

What a neutral term is can be described as a term that cannot be reduced due to it consisting of free variables. These consists of free variables themselves, applications where function is a neutral term and natural recursion where the natural number is neutral.

Below, a couple of simple properties are proved.[8]

Lemma 3.2.1. *The set of reduction rules is a subset of the conversion rules.*

Proof. Trivial, as each reduction has a conversion counterpart. □

Lemma 3.2.2. *Given any well-formed construct with a context of* Γ*, the context* Γ *is well-formed.*

Proof. Trivial, as each base case of the inductive definitions have a premise that Γ is well-formed. □

Lemma 3.2.3. *It is impossible that both* $\epsilon \vdash t : A$ *and* t *being neutral holds.*

Proof. By induction on well-formed term and neutral. □

3.2.1 Weakening

We introduce the concept of a valid weakening. This uses the predicate $\Gamma \subseteq \Delta$ which is parametrized over the two contexts Γ and Δ. The weakening here go from a context Γ and weaken it to a larger or equally large context Δ. Below are the rules for valid weakenings.[9]

$$\frac{}{\mathsf{id} : \Gamma \subseteq \Gamma} \qquad \frac{\rho : \Gamma \subseteq \Delta}{\mathsf{step}\, \rho : \Gamma \subseteq \Delta, A} \qquad \frac{\rho : \Gamma \subseteq \Delta}{\mathsf{lift}\, \rho : \Gamma, A \subseteq \Delta, \rho A}$$

Note that in the lift case, the right-hand A is weakened by ρ. In the formalisation, the weakening of the grammar and the valid weakening are defined separately, as such, a function to convert a valid weakening needs to be mutually defined. This is called toWk.

Also of note is that context members can depend on each other, as seen in the well-formed context extension. It is therefore necessary to weaken by ρ in the lift rule.

[8]These lemmas are formalised in module `Definition.Typed.Properties`

[9] The module `Definition.Typed.Weakening` of the code contains the formalisation of valid weakening and its lemmas.

Composition of weakenings in this environment is a bit more tricky. Since the propositional equality of $\mathsf{lift}\ \rho \cdot \mathsf{lift}\ \rho'$ and $\mathsf{lift}\ (\rho \cdot \rho')$ does not follow directly here, we need to prove the following lemma:

Lemma 3.2.4. *$(\rho \cdot \rho')\ t$ propositionally equal to $\rho\ (\rho'\ t)$ and composition is distributive with* toWk.

Proof. By recursion over ρ, and by using heterogeneous equality and lemma 3.1.1. □

Lemma 3.2.5 (Weakening)**.** *Given $\rho : \Gamma \subseteq \Delta$ and $\vdash \Delta$ it is such that:*

1. *If $x : A \in \Gamma$ then $\rho x : \rho A \in \Delta$.*
2. *If $\Gamma \vdash A$ then $\Delta \vdash \rho A$.*
3. *If $\Gamma \vdash t : A$ then $\Delta \vdash \rho t : \rho A$.*
4. *If $\Gamma \vdash A = B$ then $\Delta \vdash \rho A = \rho B$.*
5. *If $\Gamma \vdash t = u : A$ then $\Delta \vdash \rho t = \rho u : \rho A$.*
6. *If $\Gamma \vdash A \Rightarrow B$ then $\Delta \vdash \rho A \Rightarrow \rho B$.*
7. *If $\Gamma \vdash t \Rightarrow u : A$ then $\Delta \vdash \rho t \Rightarrow \rho u : \rho A$.*
8. *If $\Gamma \vdash A \Rightarrow^* B$ then $\Delta \vdash \rho A \Rightarrow^* \rho B$.*
9. *If $\Gamma \vdash t \Rightarrow^* u : A$ then $\Delta \vdash \rho t \Rightarrow^* \rho u : \rho A$.*

Proof. By recursion and lemmas 3.1.1 and 3.1.2.

The highlighted premises of the rules are used, as without them it would be required to use lemma 3.2.2 and pattern-match on the context. The weakening lemma would then be used on that result, however Agda cannot derive that such a type is smaller than the original type, resulting the lemma being deemed non-terminating. □

3.2.2 Reduction

For the reductions to work well with respect to computations and proofs, there are certain properties that needs to be satisfied. For instance, the reduction has to be deterministic and it has to hold that any WHNF cannot be reduced. Here the lemmas for these properties are presented.[10]

Lemma 3.2.6. *If $\Gamma \vdash t \Rightarrow u : A$ then it is impossible for t to be neutral.*

Proof. By induction on the reduction and neutral. □

Lemma 3.2.7. *If $\Gamma \vdash A \Rightarrow B$ then it is impossible for A to be of WHNF. Similarly, if $\Gamma \vdash t \Rightarrow u : C$ then it is impossible for t to be of WHNF.*

[10]These lemmas are formalised in the code in module `Definition.Typed.Properties`

Proof. By induction on the reduction and WHNF. For the application and natrec cases lemma 3.2.6 is needed. □

Lemma 3.2.8. *If $\Gamma \vdash A \Rightarrow^* B$ and A is of WHNF then A and B are propositionally equal. Similarly, if $\Gamma \vdash t \Rightarrow^* u : C$ and t is of WHNF then t and u are propositionally equal.*

Proof. By induction on the reduction closure and lemma 3.2.6 and 3.2.7. □

Lemma 3.2.9. *If $\Gamma \vdash A \Rightarrow B$ and $\Gamma \vdash A \Rightarrow B'$ then B is propositionally equal to B'. Similarly, if $\Gamma \vdash t \Rightarrow^* u : C$ and $\Gamma \vdash t \Rightarrow^* u' : C$ then u is propositionally equal to u'.*

Proof. By induction on the two reductions. For the application and natrec cases lemma 3.2.7 is needed. □

Lemma 3.2.10. *If $\Gamma \vdash A \Rightarrow^* B$ and $\Gamma \vdash A \Rightarrow^* B'$ with B and B' in WHNF then B is propositionally equal to B'. Similarly, if $\Gamma \vdash t \Rightarrow^* u : C$ and $\Gamma \vdash t \Rightarrow^* u' : C$ with u and u' in WHNF then u is propositionally equal to u'.*

Proof. By induction on the two reduction closures and lemma 3.2.7 and 3.2.9. □

4

Logical Relation

To be able to prove the properties of the language, we need a relation that reflect the semantics of the judgements. Terms of a certain type has certain properties, for instance a term of type $\mathbb{N}$ is either zero, a successor to a term of type $\mathbb{N}$ or neutral. As such a relation over the types is introduced.

We introduce *Kripke logical relations* $\Gamma \Vdash t : A$ and $\Gamma \Vdash t = t' : A$. The Kripke worlds are contexts and the Kripke relation is context extension (weakening). Our goal is to show that these relations model the syntactic relations $\Gamma \vdash t : A$ and $\Gamma \vdash t = t' : A$. In order to accomplish this we introduce a second pair of relations $\Gamma \Vdash_s t : A$ and $\Gamma \Vdash_s t = t' : A$, called *validity judgements*, that correspond to the closure of the logical relations under substitutions. We will refer to types, terms and equalities as sound if they satisfy the logical relation, and valid if they satisfy the validity judgements. The implication from $\vdash$ to $\Vdash_s$ is called the fundamental lemma of logical relations, which is the main contribution of this thesis. The general structure of fundamental and the relations can be seen in figure 4.1.

For the relations, we will use the induction-recursion pattern. This pattern works such that we inductively define one predicate and define the remaining predicates by recursion over the one inductively defined. Why this particular pattern is necessary is since Agda does not allow negative occurrences in data types, however we need to define a form of negative occurrence.

$$\begin{tikzcd}
\boxed{\Gamma \vdash A} \arrow[rr, bend left, "\text{fundamental}"] & \boxed{\Gamma \Vdash A} \arrow[l, "\text{well-formedness}"'] & \boxed{\Gamma \Vdash_s A} \arrow[l, "\text{soundness}"']
\end{tikzcd}$$

Figure 4.1: A representation of the general proof structure, where fundamental is the fundamental theorem, well-formedness is the well-formedness lemma and soundness is the logical relation projection.

4.1 Logical relation definition

For the logical relations, we will apply the induction-recursion pattern. We will define $\Gamma \Vdash A$ by induction and the other relations $\Gamma \Vdash t : A$ and $\Gamma \Vdash A = B$ and $\Gamma \Vdash t = u : A$ by recursion on the derivation $[A]$ of $\Gamma \Vdash A$. This has to be done in principle for the two levels of small types and large types. We summarize the two steps into one by parametrizing the logical relations over the level l, written $\Vdash\langle l\rangle$. The levels for small and large types are referred to as level 0 and 1 respectively.[1]

- If $\vdash \Gamma$ and there exists a type level smaller than l then $\Gamma \Vdash\langle l\rangle \mathsf{U}$ with a proof instance referred to as $[\mathsf{U}]$. If this holds, then the following holds:
  - $\Gamma \Vdash\langle l\rangle \mathsf{U} = B/[\mathsf{U}]$ if and only if U is propositionally equal to B.
  - $\Gamma \Vdash\langle l\rangle A : \mathsf{U}/[\mathsf{U}]$ if and only if $\Gamma \vdash A : \mathsf{U}$ and type level l' is smaller than l with $\Gamma \Vdash\langle l'\rangle A$.
  - $\Gamma \Vdash\langle l\rangle A = B : \mathsf{U}/[\mathsf{U}]$ if and only if $\Gamma \vdash A : \mathsf{U}$ and $\Gamma \vdash B : \mathsf{U}$ and $\Gamma \vdash A = B : \mathsf{U}$ and l' is a smaller type level than l with $\Gamma \Vdash\langle l'\rangle A$ and $\Gamma \Vdash\langle l'\rangle B$ and $\Gamma \Vdash\langle l'\rangle A = B$.
- If we have $\Gamma \vdash A \Rightarrow^* \mathbb{N}$ then we have $\Gamma \Vdash\langle l\rangle A$ Given we have instance $[A]$ of that particular $\Gamma \Vdash\langle l\rangle A$, the following also holds:
  - $\Gamma \Vdash\langle l\rangle A = B/[A]$ if and only if $\Gamma \vdash B \Rightarrow^* \mathbb{N}$.
  - $\Gamma \Vdash\langle l\rangle t : A/[A]$ if and only if $\Gamma \vdash t :\Rightarrow^*: n : \mathbb{N}$ where n is either zero, a successor of some term or neutral, and the following has to hold:
    * If n is a successor of m, then $\Gamma \Vdash\langle l\rangle m : \mathbb{N}$.
    * If n is neutral, then $\Gamma \vdash n : \mathbb{N}$.
  - $\Gamma \Vdash\langle l\rangle t = u : A/[A]$ if and only if $\Gamma \vdash t \Rightarrow^* n : \mathbb{N}$ and $\Gamma \vdash u \Rightarrow^* m : \mathbb{N}$ where n and m are either both zero, successor to some term or neutral, and the following has to hold:
    * If n and m are successors to n' and m' respectively, then the equality $\Gamma \Vdash\langle l\rangle n' = m' : \mathbb{N}$ has to hold.
    * If n and m are neutrals, then $\Gamma \vdash n = m : \mathbb{N}$.
- If we have $\Gamma \vdash A \Rightarrow^* K$ with K being neutral then we have $\Gamma \Vdash\langle l\rangle A$. Given we have instance $[A]$ of that particular $\Gamma \Vdash\langle l\rangle A$, the following also holds:
  - $\Gamma \Vdash\langle l\rangle A = B/[A]$ if and only if $\Gamma \vdash B \Rightarrow^* M$ with M neutral and $\Gamma \vdash K = M$.
  - $\Gamma \Vdash\langle l\rangle t : A/[A]$ if and only if $\Gamma \vdash t : A$.
  - $\Gamma \Vdash\langle l\rangle t = u : A/[A]$ if and only if $\Gamma \vdash t = u : A$.

[1]In the code, the logical relation is defined in the module `Definition.LogicalRelation`.

- If we have $\Gamma \vdash A \Rightarrow^* \Pi\, F\, G$ and $\Gamma \vdash F$ and $\Gamma, F \vdash G$ as well as the following properties given any $\rho : \Gamma \subseteq \Delta$ and $\vdash \Delta$:
  - $\Delta \Vdash\langle l\rangle\ \rho F$ has to hold. We refer to it here as $[F]$.
  - Given $\Delta \Vdash\langle l\rangle\ a : \rho F/[F]$ we have $\Delta \Vdash\langle l\rangle\ (\mathsf{lift}\,\rho)G[a]$. We refer to it here as $[G]$.
  - Given $\Delta \Vdash\langle l\rangle\ a : \rho F/[F]$ and $\Delta \Vdash\langle l\rangle\ b : \rho F/[F]$ and $\Delta \Vdash\langle l\rangle\ a = b : \rho F$ $/[F]$ then we have $\Delta \Vdash\langle l\rangle\ (\mathsf{lift}\,\rho)G[a] = (\mathsf{lift}\,\rho)G[b]/[G]$.

  When we have all this we have $\Gamma \Vdash\langle l\rangle\ A$. For any instance $[A]$ of $\Gamma \Vdash\langle l\rangle\ A$ where the above holds, the following also holds:
  - $\Gamma \Vdash\langle l\rangle\ A = B/[A]$ if and only if the following: Suppose that we have $\Gamma \vdash B \Rightarrow^* \Pi\, F'\, G'$ such that given $\rho : \Gamma \subseteq \Delta$ and $\vdash \Delta$ we have both $\Delta \Vdash\langle l\rangle\ \rho F = \rho F'/[F]$ and for all $\Delta \Vdash\langle l\rangle\ a : \rho F/[F]$ we then have $\Delta \Vdash\langle l\rangle\ (\mathsf{lift}\,\rho)G[a] = (\mathsf{lift}\,\rho)G'[a]/[G]$.
  - $\Gamma \Vdash\langle l\rangle\ t : A/[A]$ if and only if we have the following: We have $\Gamma \vdash t : A$ and given $\rho : \Gamma \subseteq \Delta$ and $\vdash \Delta$ and $\Delta \Vdash\langle l\rangle\ a : \rho F/[F]$ then we have $\Delta \Vdash\langle l\rangle\ (\rho t)\, a : (\mathsf{lift}\,\rho)G[a]\ /\ [G]$ and for all $\Delta \Vdash\langle l\rangle\ b : \rho F\ /\ [F]$ and $\Delta \Vdash\langle l\rangle\ a = b : \rho F/[F]$ we have $\Delta \Vdash\langle l\rangle\ (\rho t)\, a = (\rho t)\, b : (\mathsf{lift}\,\rho)G[a]/[G]$.
  - $\Gamma \Vdash\langle l\rangle\ t = u : A/[A]$ if and only if the following holds: We have $\Gamma \vdash t = u : A$ and $\Gamma \Vdash\langle l\rangle\ t : A/[A]$ and $\Gamma \Vdash\langle l\rangle\ u : A/[A]$ and given $\rho : \Gamma \subseteq \Delta$ and $\vdash \Delta$ and $\Delta \Vdash\langle l\rangle\ a : \rho F/[F]$ then we have $\Delta \Vdash\langle l\rangle\ (\rho t)\, a = (\rho u)\, a : (\mathsf{lift}\,\rho)G[a]/[G]$.
- If we have a derivation $[A]'$ of $\Gamma \Vdash\langle l'\rangle\ A$ and l' is a smaller type level than l, then $\Gamma \Vdash\langle l\rangle\ A$ and let $[A]$ be a derivation of it. Additionally, the following holds from this:
  - $\Gamma \Vdash\langle l\rangle\ A = B/[A]$ if and only if $\Gamma \Vdash\langle l'\rangle\ A = B/[A]'$.
  - $\Gamma \Vdash\langle l\rangle\ t : A/[A]$ if and only if $\Gamma \Vdash\langle l'\rangle\ t : A/[A]'$.
  - $\Gamma \Vdash\langle l\rangle\ t = u : A/[A]$ if and only if $\Gamma \Vdash\langle l'\rangle\ t = u : A/[A]'$.

4.1.1 Equality view

When pattern matching, Agda can automatically refute cases if there is an impossibility. However, this is only if the impossibility arises without further normalisation.

For the types of the logical relation, it is not immediately obvious that two sound types with different outermost type constructor cannot be reducts of the same type. For instance, if A both reduces to $\mathbb{N}$ and $\Pi\, F\, G$ for any F and G, we arrive at an impossibility since both $\mathbb{N}$ and $\Pi\, F\, G$ are of WHNF as per lemma 3.2.10.

In some of the following lemmas, there are premises $\Gamma \Vdash\langle l\rangle\ A$ with derivation $[A]$ and $\Gamma \Vdash\langle l\rangle\ B$ and $\Gamma \Vdash\langle l\rangle\ A = B/[A]$ where it is necessary to recurse over the types. As there are five different sound type constructors, we end up with 25 cases, or 18 by not using recursion for the first instance when the other is an embedding construct. For the remaining cases, 12 of the 18 can be refuted.

To avoid repeating the refuting cases for each lemma, a view of two sound types is introduced without these cases. The view then consists of when both types are U, reduce to $\mathbb{N}$, reduce to two possibly different neutrals, reduce to $\Pi\, F\, G$ and $\Pi\, F'\, G'$ or one of the sound types is an embedding.[2]

Lemma 4.1.1. *If* $[A] : \Gamma \Vdash\langle l\rangle\ A$ *and* $[B] : \Gamma \Vdash\langle l\rangle\ B$ *and* $\Gamma \Vdash\langle l\rangle\ A = B/[A]$ *then the equality view of* $[A]$ *and* $[B]$ *holds.*

Proof. By recursion on $[A]$ and $[B]$, the cases in the view are trivial and the cases not in the view are refuted by lemma 3.2.10. □

For some of the following lemmas we also need a variant where A and B are the same. Therefore the reflexivity lemma is introduced:

Lemma 4.1.2 (Reflexivity)**.** *For all* $[A] : \Gamma \Vdash\langle l\rangle\ A$ *it follows that* $\Gamma \Vdash\langle l\rangle\ A = A/[A]$*. Additionally, if* $\Gamma \Vdash\langle l\rangle\ t : A/[A]$ *then* $\Gamma \Vdash\langle l\rangle\ t = t : A/[A]$*.*

Proof. By induction on the sound type or term. □

Corollary 4.1.3. *If there is two instances* $[A]$ *and* $[A]'$ *of* $\Gamma \Vdash\langle l\rangle\ A$ *and* $\Gamma \Vdash\langle l'\rangle\ A$ *respectively, then there is an equality view of* $[A]$ *and* $[A]'$*.*

4.1.2 Proof irrelevance

As the logical relation uses induction-recursion, the recursions depend on the induction. This means that if we let $[A]$ and $[A]'$ be of type $\Gamma \Vdash A$ then $\Gamma \Vdash A = B/[A]$ is not the same as $\Gamma \Vdash A = B/[A]'$. The same applies to the other recursive instances.

Here we prove that we can change the sound type instance of the recursion instances. This lemma is particularly useful when we have two types which are propositionally equal to each other, A and A', and we need to show that a term is a sound member of A when it is a sound member of A'.[3]

Lemma 4.1.4 (Proof irrelevance)**.** *If* $[A] : \Gamma \Vdash\langle l\rangle\ A$ *and* $[A]' : \Gamma \Vdash\langle l'\rangle\ A$ *then:*

1. *If* $\Gamma \Vdash\langle l\rangle\ A = B/[A]$ *then* $\Gamma \Vdash\langle l'\rangle\ A = B/[A]'$*.*
2. *If* $\Gamma \Vdash\langle l\rangle\ t : A/[A]$ *then* $\Gamma \Vdash\langle l'\rangle\ t : A/[A]'$*.*
3. *If* $\Gamma \Vdash\langle l\rangle\ t = u : A/[A]$ *then* $\Gamma \Vdash\langle l'\rangle\ t = u : A/[A]'$*.*

Proof. The equality view and lemma 4.1.3 is used to refute the refutable cases. Most of the other cases are trivial. For the neutral case in (1) lemma 3.2.1 is used. For cases of Π type, recursion is used for the weakening properties. □

[2]In the code this is known as `Tactic` from the module `Definition.LogicalRelation.Tactic`.
[3] This is formally proven in the module `Definition.LogicalRelation.Irrelevance`.

With the proof irrelevance lemma, we can assert that for any recursion instance of the induction-recursion definition, we do not need a specific instance of a proof, but only that there exists a proof. Hence if $\Gamma \Vdash\langle l\rangle\ A$ then $\Gamma \Vdash\langle l\rangle\ A = B/[A]$ can be simplified to $\Gamma \Vdash\langle l\rangle\ A = B$. Here we will also leave out the type level when it can be any value. Thus the following simplifications are done:

- Soundness of type equality is simplified to $\Gamma \Vdash A = B$.
- Soundness of a term is simplified to $\Gamma \Vdash t : A$.
- Soundness of term equality is simplified to $\Gamma \Vdash t = u : A$.

Similarly to the judgements, we also introduce $\Gamma \Vdash A :=: B$ and $\Gamma \Vdash t :=: u : A$ for cases which not only equality needs to be sound, but also the types and terms.

$$\frac{\Gamma \Vdash A \qquad \Gamma \Vdash B \qquad \Gamma \Vdash A = B}{\Gamma \Vdash A :=: B} \qquad \frac{\Gamma \Vdash t : A \qquad \Gamma \Vdash u : A \qquad \Gamma \Vdash t = u : A}{\Gamma \Vdash t :=: u : A}$$

4.1.3 Language properties

Here lemmas are presented for showing that sound constructs are also well-formed, as well as properties following the judgements are still holding.[4]

Lemma 4.1.5 (Well-formedness)**.** *Any sound construct is also well-formed, that is if* $\Gamma \Vdash A$ *then* $\Gamma \vdash A$*. Additionally the following holds:*

- *If* $\Gamma \Vdash A = B$ *then* $\Gamma \vdash A = B$.
- *If* $\Gamma \Vdash t : A$ *then* $\Gamma \vdash t : A$.
- *If* $\Gamma \Vdash t = u : A$ *then* $\Gamma \vdash t = u : A$.

Proof. By induction on the sound construct and lemma 3.2.1. □

Lemma 4.1.6 (Weakening)**.** *If* $\rho : \Gamma \subseteq \Delta$ *and* $\vdash \Delta$ *and* $\Gamma \Vdash\langle l\rangle\ A$ *then* $\Delta \Vdash\langle l\rangle\ \rho A$*. Additionally:*

- *If* $\Gamma \Vdash\langle l\rangle\ A = B$ *then* $\Delta \Vdash\langle l\rangle\ \rho A = \rho B$.
- *If* $\Gamma \Vdash\langle l\rangle\ t : A$ *then* $\Delta \Vdash\langle l\rangle\ \rho t : \rho A$.
- *If* $\Gamma \Vdash\langle l\rangle\ t = u : A$ *then* $\Delta \Vdash\langle l\rangle\ \rho t = \rho u : \rho A$.

Proof. By induction on the sound construct. For most cases we simply use lemma 3.2.5. For the Π cases, lemmas 3.2.4 and 4.1.4 are used. □

[4] These lemmas are formalised in modules `Definition.LogicalRelation.Weakening` and `Definition.LogicalRelation.Properties`.

Lemma 4.1.7 (Type conversion)**.** *If* $\Gamma \Vdash A = B$ *then:*

- $\Gamma \Vdash t : A$ *if and only if* $\Gamma \Vdash t : B$.
- $\Gamma \Vdash t = u : A$ *if and only if* $\Gamma \Vdash t = u : B$.

Proof. By induction on the sound term or equality using the equality view with lemma 4.1.1. □

Lemma 4.1.8 (Symmetry)**.** *If* $\Gamma \Vdash A = B$ *then* $\Gamma \Vdash B = A$*. Additionally, if* $\Gamma \Vdash t = u : A$ *then* $\Gamma \Vdash u = t : A$.

Proof. By induction on the sound equality using the equality view with lemma 4.1.1. For the Π case lemma 4.1.7 is needed. □

Lemma 4.1.9 (Transitivity)**.** *If* $\Gamma \Vdash A = B$ *and* $\Gamma \Vdash B = C$ *then* $\Gamma \Vdash A = C$*. Additionally, if* $\Gamma \Vdash t = u : A$ *and* $\Gamma \Vdash u = r : A$ *then* $\Gamma \Vdash t = r : A$.

Proof. By induction on the sound equality using the equality view with lemma 4.1.1. For the Π case lemma 4.1.7 is needed. □

Lemma 4.1.10 (Natural soundness)**.** *For all* Γ *which are well-formed,* $\Gamma \Vdash \mathbb{N}$ *and* $\Gamma \Vdash \mathsf{zero} : \mathbb{N}$ *holds. Additionally, if* $\Gamma \Vdash n : \mathbb{N}$ *then* $\Gamma \Vdash \mathsf{suc}\, n : \mathbb{N}$*. Similarly, if* $\Gamma \Vdash n = n' : \mathbb{N}$ *then* $\Gamma \Vdash \mathsf{suc}\, n = \mathsf{suc}\, n' : \mathbb{N}$.

Proof. By induction on the sound term or equality. □

Lemma 4.1.11 (Neutral soundness)**.** *The following properties holds:*

- *If* $\Gamma \vdash A$ *and* A *is neutral, then* $\Gamma \Vdash A$.
- *If* $\Gamma \vdash A :=: B$ *and both* A *and* B *are neutral, then* $\Gamma \Vdash A = B$.
- *If* $\Gamma \Vdash A$ *and* $\Gamma \vdash t : A$ *and* t *is neutral, then* $\Gamma \Vdash t : A$.
- *If* $\Gamma \Vdash A$ *and* $\Gamma \vdash t :=: u : A$ *and both* t *and* u *is neutral, then* $\Gamma \Vdash t = u : A$.

Proof. By induction on the type. For the Π cases lemma 3.2.1 and 4.1.5 are needed. □

Lemma 4.1.12 (Reduction soundness)**.** *The following properties holds:*

- *If* $\Gamma \vdash A \Rightarrow B$ *and* $\Gamma \Vdash B$ *then* $\Gamma \Vdash A$ *and* $\Gamma \Vdash A = B$.
- *If* $\Gamma \vdash t \Rightarrow u : A$*,* $\Gamma \Vdash A$ $\Gamma \Vdash u : A$ *then* $\Gamma \Vdash t : A$ *and* $\Gamma \Vdash t = u : A$.
- *If* $\Gamma \vdash A \Rightarrow^* B$ *and* $\Gamma \Vdash B$ *then* $\Gamma \Vdash A$ *and* $\Gamma \Vdash A = B$.
- *If* $\Gamma \vdash t \Rightarrow^* u : A$*,* $\Gamma \Vdash A$ $\Gamma \Vdash u : A$ *then* $\Gamma \Vdash t : A$ *and* $\Gamma \Vdash t = u : A$.

Proof. By induction on the type. For the Π cases lemma 3.2.1, 4.1.5, 4.1.7, 4.1.8 and 4.1.9 are needed. □

Lemma 4.1.13 (Application soundness)**.** *Whenever* $\Gamma \Vdash F$ *and* $\Gamma \Vdash G[u]$ *and* $\Gamma \Vdash \Pi F\, G$ *and* $\Gamma \Vdash t : \Pi F\, G$ *and* $\Gamma \Vdash u : F$ *then* $\Gamma \Vdash t\, u : G[u]$*. Similarly, whenever* $\Gamma \Vdash F$ *and* $\Gamma \Vdash G[u]$ *and* $\Gamma \Vdash \Pi F\, G$ *and* $\Gamma \Vdash t = t' : \Pi F\, G$ *and* $\Gamma \Vdash u :=: u' : F$ *then* $\Gamma \Vdash t\, u = t'\, u' : G[u]$*.*

Proof. By lemmas 3.2.2, 3.2.8 and 4.1.4. □

4.2 Validity judgements

For the validity judgements, we introduce the concepts of context and substitution validity as well as the type, term and their respective equalities. Just as in the logical relation, induction-recursion is used where valid contexts are defined by induction and the rest are defined by recursion on valid contexts.[5]

A context is valid if it is either empty or if it extends a valid context with a type that is valid in the validity judgements. As such, the definitions of valid types and contexts are mutually dependent. Formally we denote a context Γ valid as $\Vdash_s \Gamma$.

A substitution σ is valid given we have a valid context Γ and a well-formed context Δ as well as some additional requirements based on Γ. Γ is here the context which an expression is valid in to be able to apply σ and Δ is the context which the resulting application is valid in. Given that $[\Gamma]$ is an instance of $\Vdash_s \Gamma$ and $[\Delta]$ is an instance of $\vdash \Delta$ substitution validity is formally denoted as $\Delta \Vdash_s \sigma : \Gamma / [\Gamma] / [\Delta]$ with $[\sigma]$ as an instance of it, and $\Delta \Vdash_s \sigma = \sigma' : \Gamma / [\Gamma] / [\Delta] / [\sigma]$ for validity of substitution equality. The additional requirements are defined recursively over Γ:

- If Γ is an empty context, then any σ is valid and the equality is valid for any σ and σ'. Since there is nothing to substitute in an expression valid in an empty context, any substitution is effectively an identity substitution.
- Otherwise, Γ extends a smaller context E by type A. σ is valid if $\Delta \Vdash_s \mathsf{tail}\, \sigma : E / [E] / [\Delta]$ and $\Delta \Vdash \mathsf{head}\, \sigma : (\mathsf{tail}\, \sigma) A / [A]$. Equality of σ and σ' is valid if $\Delta \Vdash_s \mathsf{tail}\, \sigma = \mathsf{tail}\, \sigma' : E / [E] / [\Delta] / [\sigma]$ and $\Delta \Vdash \mathsf{head}\, \sigma = \mathsf{head}\, \sigma' : (\mathsf{tail}\, \sigma) A / [A]$ as well as $\Delta \Vdash_s \sigma : \Gamma / [\Gamma] / [\Delta]$.

For the typing and term membership validity, it has to hold that the context Γ is also valid with instance $[\Gamma]$. Additionally, the following has to hold for all well-formed contexts Δ with instance $[\Delta]$ and substitutions $[\sigma] : \Delta \Vdash_s \sigma : \Gamma / [\Gamma] / [\Delta]$:

- A type A is valid if $\Delta \Vdash \langle l \rangle\ \sigma A$ is satisfied and for all σ' with $\Delta \Vdash_s \sigma = \sigma' : \Gamma / [\Gamma] / [\Delta] / [\sigma]$ if $\Delta \Vdash \langle l \rangle\ \sigma A = \sigma' A$ holds. It is formally denoted as $\Gamma \Vdash_s \langle l \rangle\ A / [\Gamma]$.
- Equality of types A and B is valid given A is a valid type with instance $[A]$. It also has to satisfy $\Delta \Vdash \langle l \rangle\ \sigma A = \sigma B$. It is formally denoted as $\Gamma \Vdash_s \langle l \rangle\ A = B / [\Gamma] / [A]$.

[5] The module `Definition.LogicalRelation.Substitution` contains the definitions of the validity judgements.

- A term t of type A is valid given that A is a valid type with instance $[A]$. Additionally, it is required that $\Delta \Vdash \sigma t : \sigma A$ holds and for all σ' satisfying $\Delta \Vdash_s \sigma = \sigma' : \Gamma$ it requires that $\Delta \Vdash\langle l\rangle\ \sigma t = \sigma' t : \sigma A$ is satisfied. It is formally denoted as $\Gamma \Vdash_s\langle l\rangle\ t : A/[\Gamma]/[A]$.
- Equality of terms t and u of type A is valid given A is a valid type with instance $[A]$. It also has to satisfy that $\Delta \Vdash \sigma t = \sigma u : \sigma A$ holds. It is formally denoted as $\Gamma \Vdash_s\langle l\rangle\ t = u : A/[\Gamma]/[A]$.

Additionally, given derivation $[\Gamma]$ of $\Vdash_s \Gamma$, validity of term reduction $\Gamma \Vdash_s t \Rightarrow u : A$ $/[\Gamma]$ holds if and only if given derivation $[\Delta]$ of $\vdash \Delta$ and $\Delta \Vdash_s \sigma : \Gamma/[\Gamma]/[\Delta]$ such that $\Delta \vdash \sigma t \Rightarrow \sigma u : \sigma A$ holds.

We can see in figure 4.2 that a lot of the parts of the validity judgements mutually depend on each others.

$$\begin{tikzcd}
 & \boxed{\Gamma \Vdash_s A} \arrow[d, bend left] & \\
 & \boxed{\Vdash_s \Gamma} \arrow[u, bend left] \arrow[dl] \arrow[dr] & \\
\boxed{\Delta \Vdash_s \sigma : \Gamma} \arrow[uur, bend left] \arrow[rr, bend left] & & \boxed{\Delta \Vdash_s \sigma = \sigma' : \Gamma} \arrow[uul, bend right] \arrow[ll, bend left]
\end{tikzcd}$$

Figure 4.2: Graph of the definition dependencies of the validity judgements. Type equality, type membership and term equality are omitted to make it easier to read.

4.2.1 Validity of substitutions

Here a couple of lemmas for valid substitutions are presented. These lemmas cover the irrelevance of inductive proofs of the inductive-recursive relation as well as weakening and lifting of valid substitutions and substitution equalities. Additionally, the reflexivity, symmetry and transitivity properties are covered for substitution equality.[6]

Lemma 4.2.1 (Substitution irrelevance). *Given two instances* $[\Gamma]$ *and* $[\Gamma]'$ *of* $\Vdash_s \Gamma$ *and two instances* $[\Delta]$ *and* $[\Delta]'$ *of* $\vdash \Delta$ *then whenever* $[\sigma] : \Delta \Vdash_s \sigma : \Gamma/[\Gamma]/[\Delta]$ *holds,*

[6]The module `Definition.LogicalRelation.Substitution.Irrelevance` contains the formalisation of the irrelevance lemma and the remaining lemmas are formalised in module `Definition.LogicalRelation.Substitution.Properties`.

then $[\sigma]' : \Delta \Vdash_s \sigma : \Gamma/[\Gamma]'/[\Delta]'$ *also holds. Additionally, if* $\Delta \Vdash_s \sigma = \sigma' : \Gamma/[\Gamma]/[\Delta]$ $/[\sigma]$ *holds, then* $\Delta \Vdash_s \sigma = \sigma' : \Gamma/[\Gamma]'/[\Delta]'/[\sigma]'$ *also holds.*

Proof. By induction on the valid substitution and lemma 4.1.4. □

With proof irrelevance for the substitution validity, we can now assert that it is only necessary that there exists a proof and does not have to be a specific instance. As such, the following simplifications are made:

- Valid substitution is simplified to $\Delta \Vdash_s \sigma : \Gamma$.
- Equality of valid substitutions is simplified to $\Delta \Vdash_s \sigma = \sigma' : \Gamma$.

Lemma 4.2.2. *Given that a context* Γ *is valid, it is also well-formed as well as* $\Gamma \Vdash_s \mathsf{id} : \Gamma$ *holding.*

Proof. By induction on the valid context and lemma 4.1.5. The two conclusions are mutually dependent. □

Lemma 4.2.3 (Substitution weakening)**.** *Given a valid context* Γ *and well-formed contexts* Δ *and* E*, if* $\rho : \Delta \subseteq E$ *and* $\Delta \Vdash_s \sigma : \Gamma$ *then* $E \Vdash_s \rho\sigma : \Gamma$*. Additionally, if* $\Delta \Vdash_s \sigma = \sigma' : \Gamma$ *then* $E \Vdash_s \rho\sigma = \rho\sigma' : \Gamma$*.*

Proof. By induction on the valid substitution and lemma 4.1.6 and 3.1.3. □

Lemma 4.2.4 (Substitution lifting)**.** *Given a valid context* Γ *with instance* $[\Gamma]$ *and a well-formed context* Δ*, if* $\Gamma \Vdash_s\langle l\rangle\ F/[\Gamma]$ *and* $\Delta \Vdash_s \sigma : \Gamma$ *then* $\Delta, \sigma F \Vdash_s \mathsf{lift}\,\sigma : \Gamma, F$*. Additionally, if* $\Delta \Vdash_s \sigma = \sigma' : \Gamma$ *then* $\Delta, \sigma F \Vdash_s \mathsf{lift}\,\sigma = \mathsf{lift}\,\sigma' : \Gamma, F$*.*

Proof. By lemma 3.1.3, 4.1.5, 4.1.11 and 4.2.3. □

Lemma 4.2.5. *For the equality of valid substitutions, the reflexivity, symmetry and transitivity properties holds.*

Proof. By the respective lemmas 4.1.2, 4.1.8 and 4.1.9 as well as lemma 4.1.7. □

4.2.2 Validity of types and terms

In this section, proofs for various type, term and equality introductions and substitutions. Similarly to the logical relation, the proof irrelevance property is proved first to simplify the validity judgements.[7]

Lemma 4.2.6 (Proof irrelevance)**.** *Given the two instances* $[\Gamma]$ *and* $[\Gamma]'$ *of* $\Vdash_s \Gamma$ *and* $[A] : \Gamma \Vdash_s\langle l\rangle\ A/[\Gamma]$ *then* $\Gamma \Vdash_s\langle l\rangle\ A/[\Gamma]'$*. Additionally if* $[A]' : \Gamma \Vdash_s\langle l'\rangle\ A/[\Gamma]'$ *then:*

- *Given* $\Gamma \Vdash_s\langle l\rangle\ A = B/[\Gamma]/[A]$ *then* $\Gamma \Vdash_s\langle l'\rangle\ A = B/[\Gamma]'/[A]'$*.*
- *Given* $\Gamma \Vdash_s\langle l\rangle\ t : A/[\Gamma]/[A]$ *then* $\Gamma \Vdash_s\langle l'\rangle\ t : A/[\Gamma]'/[A]'$*.*

[7] In the module `Definition.LogicalRelation.Substitution.Irrelevance` the irrelevance lemma is formalised.

- *Given* $\Gamma \Vdash_s \langle l \rangle \; t = u : A / [\Gamma] / [A]$ *then* $\Gamma \Vdash_s \langle l' \rangle \; t = u : A / [\Gamma]' / [A]'$.

Proof. By induction on the valid construct and lemma 4.1.4 and 4.2.1. □

Similarly as in the logical relation, with the irrelevance lemma, we can now assert that dependency of specific proofs and type levels for logical relation instances can be omitted. As such we have the following simplifications:

- Valid types are denoted as $\Gamma \Vdash_s A$.
- Valid equality of types are denoted as $\Gamma \Vdash_s A = B$.
- Valid term of a type are denoted as $\Gamma \Vdash_s t : A$.
- Valid equality of terms of a type are denoted as $\Gamma \Vdash_s t = u : A$.

Similar to the judgements and the logical relation, we introduce $\Gamma \Vdash_s A :=: B$ and $\Gamma \Vdash_s t :=: u : A$ for when a lemma not only need a valid equality, but also that the types and terms are valid.

$$\frac{\Gamma \Vdash_s A \qquad \Gamma \Vdash_s B \qquad \Gamma \Vdash_s A = B}{\Gamma \Vdash_s A :=: B} \qquad \frac{\Gamma \Vdash_s t : A \qquad \Gamma \Vdash_s u : A \qquad \Gamma \Vdash_s t = u : A}{\Gamma \Vdash_s t :=: u : A}$$

Following are the lemmas to prove the validity of different types and terms.[8]

Lemma 4.2.7 (Universe validity). *If* $\Vdash_s \Gamma$ *then* $\Gamma \Vdash_s \mathsf{U}$. *Additionally, if* $\Gamma \Vdash_s A : \mathsf{U}$ *then* $\Gamma \Vdash_s A$. *Similarly, if* $\Gamma \Vdash_s A = B : \mathsf{U}$ *then* $\Gamma \Vdash_s A = B$.

Lemma 4.2.8 (Natural validity). *If* $\Vdash_s \Gamma$ *then* $\Gamma \Vdash_s \mathbb{N}$ *and* $\Gamma \Vdash_s \mathsf{zero} : \mathbb{N}$. *Additionally, if* $\Gamma \Vdash_s n : \mathbb{N}$ *then* $\Gamma \Vdash_s \mathsf{suc}\, n : \mathbb{N}$.

Lemma 4.2.9 (Type conversion). *If* $\Vdash_s \Gamma$ *and* $\Gamma \Vdash_s A :=: B$ *then the following holds:*

- $\Gamma \Vdash_s t : A$ *if and only if* $\Gamma \Vdash_s t : B$.
- $\Gamma \Vdash_s t = u : A$ *if and only if* $\Gamma \Vdash_s t = u : B$.

Lemma 4.2.10 (Single substitution validity). *Given valid context* Γ *and* $\Gamma \Vdash_s F$ *and* $\Gamma, F \Vdash_s G$ *and* $\Gamma \Vdash_s t : F$ *then* $\Gamma \Vdash_s G[t]$. *Also following this:*

- *If* $\Gamma, F \Vdash_s g : G$ *then* $\Gamma \Vdash_s g[t] : G[t]$.
- *If* $\Gamma \Vdash_s F :=: F'$ *and* $\Gamma, F \Vdash_s G$ *and* $\Gamma, F' \Vdash_s G$ *and* $\Gamma, F \Vdash_s G = G'$ *and* $\Gamma \Vdash_s t : F$ *and* $\Gamma \Vdash_s t' : F'$ *and* $\Gamma \Vdash_s t = t' : F$ *then* $\Gamma \Vdash_s G[t] = G'[t']$.

Lemma 4.2.11 (Lifting single substitution validity). *If* $\Vdash_s \Gamma$ *and* $\Gamma \Vdash_s F$ *and* $\Gamma, F \Vdash_s G$ *and* $\Gamma \Vdash_s t : \mathsf{wk1}\, F$ *then* $\Gamma, F \Vdash_s G[t]\!\uparrow$. *Additionally, if* $\Gamma, F \Vdash_s G :=: G'$ *and* $\Gamma, F \Vdash_s t :=: t' : \mathsf{wk1}\, F$ *then* $\Gamma, F \Vdash_s G[t]\!\uparrow = G'[t']\!\uparrow$.

[8]In module `Definition.LogicalRelation.Substitution.Introductions` these lemmas are formalised.

Lemma 4.2.12 (Π-type validity). *Given* $\Vdash_s \Gamma$ *and* $\Gamma \Vdash_s F$ *and* $\Gamma, F \Vdash_s G$ *then* $\Gamma \Vdash_s \Pi\, F\, G$. *Additionally, if also* $\Gamma \Vdash_s H$ *and* $\Gamma, H \Vdash_s E$ *and* $\Gamma \Vdash_s F = H$ *and* $\Gamma, F \Vdash_s G = E$ *holds, then* $\Gamma \Vdash_s \Pi\, F\, G = \Pi\, H\, E$.

Lemma 4.2.13 (Application validity). *Given* $\Vdash_s \Gamma$ *and* $\Gamma \Vdash_s F$ *and* $\Gamma \Vdash_s \Pi\, F\, G$ *and* $\Gamma \Vdash_s t : \Pi\, F\, G$ *and* $\Gamma \Vdash_s u : F$ *then* $\Gamma \Vdash_s t\, u : G[u]$. *Similarly for congruence, if also* $\Gamma \Vdash_s t = t' : \Pi\, F\, G$ *and* $u \Vdash_s F :=: u' : F$ *holds, then* $\Gamma \Vdash_s t\, u = t'\, u' : G[u]$.

Lemma 4.2.14 (λ validity). *Given valid context* Γ *and* $\Gamma \Vdash_s F$ *and* $\Gamma, F \Vdash_s G$ *and* $\Gamma, F \Vdash_s t : G$ *then* $\Gamma \Vdash_s \lambda\, t : \Pi\, F\, G$.

Lemma 4.2.15. *Given* $\Vdash_s \Gamma$ *and* $\Gamma \Vdash_s F$ *and* $\Gamma, F \Vdash_s G$ *and* $\Gamma \Vdash_s f : \Pi\, F\, G$ *and* $\Gamma \Vdash_s g : \Pi\, F\, G$ *and* $\Gamma, F \Vdash_s \mathsf{wk1}\, f\, x = \mathsf{wk1}\, g\, x : G$ *then* $\Gamma \Vdash_s f = g : \Pi\, F\, G$.

Lemma 4.2.16 (Reduction validity). *Given* $\Vdash_s \Gamma$ *and* $\Gamma \Vdash_s A$ *and* $\Gamma \Vdash_s t \Rightarrow u : A$ *and* $\Gamma \Vdash_s u : A$ *then* $\Gamma \Vdash_s t : A$ *and* $\Gamma \Vdash_s t = u : A$.

Lemma 4.2.17 (Natural recursion validity). *Given valid context* Γ *and* $\Gamma, \mathbb{N} \Vdash_s F$ *and* $\Gamma \Vdash_s F[\mathsf{zero}]$ *and* $\Gamma \Vdash_s \Pi\, \mathbb{N}\, (F \to F[\mathsf{suc}\, \mathsf{x_0}]\!\uparrow)$ *and* $\Gamma \Vdash_s F[n]$ *and* $\Gamma \Vdash_s z : F[\mathsf{zero}]$ *and* $\Gamma \Vdash_s s : \Pi\, \mathbb{N}\, (F \to F[\mathsf{suc}\, \mathsf{x_0}]\!\uparrow)$ *and* $\Gamma \Vdash_s n : \mathbb{N}$ *then* $\Gamma \Vdash_s \mathsf{natrec}\, F\, z\, s\, n : F[n]$

Lemma 4.2.18 (Well-formedness). *Given a valid type, term or respective equality, it is also well-formed.*

4.3 Fundamental theorem

With the logical relation and validity judgements defined, and their lemmas proven, it is now possible to prove the fundamental theorem.[9]

Theorem 4.3.1 (Fundamental). *Given any well-formed context, type, term or equality of types or terms, it is also valid. More specifically:*

1. *If* $\vdash \Gamma$ *then* $\Vdash_s \Gamma$.
2. *If* $\Gamma \vdash A$ *then* $\Vdash_s \Gamma$ *and* $\Gamma \Vdash_s A$.
3. *If* $\Gamma \vdash A = B$ *then* $\Vdash_s \Gamma$ *and* $\Gamma \Vdash_s A$ *and* $\Gamma \Vdash_s B$.
4. *If* $\Gamma \vdash t = u : A$ *then* $\Vdash_s \Gamma$ *and* $\Gamma \Vdash_s A$ *and* $\Gamma \Vdash_s t : A$ *and* $\Gamma \Vdash_s u : A$ *and* $\Gamma \Vdash_s t = u : A$.

This theorem states that any well-formed instance will fulfil the logical relations properties. This means that many more properties can be derived from a well-formed instance, which lets us prove further properties of the language itself.

[9] The module `Definition.LogicalRelation.Fundamental` contains the formalisation of the fundamental lemma.

4.3.1 Consequences

With the fundamental theorem, Π-injectivity can be proven along with other theorems. These theorems are of the form of taking a judgement and showing that an other judgement is valid.[10]

Theorem 4.3.2 (Π-injectivity)**.** *Given* $\Gamma \vdash \Pi\, F\, G = \Pi\, H\, E$ *then* $\Gamma \vdash F = H$ *and* $\Gamma, F \vdash G = E$ *holds.*

Proof. By theorem 4.3.1 and lemma 4.1.11. □

Theorem 4.3.3 (Weak canonicity)**.** *Given* $\epsilon \vdash n : \mathbb{N}$ *then there exists a* k *such that* $\epsilon \vdash n = \mathsf{suc}^k\, \mathsf{zero} : \mathbb{N}$ *where* $\mathsf{suc}^k\, \mathsf{zero}$ *is* suc *applied* k *times to* zero*.*

Proof. By theorem 4.3.1 and induction on the sound n. The neutral case is refuted by lemma 3.2.3. For the $\mathsf{suc}\, m$ case we use recursion on m and for both $\mathsf{suc}\, m$ and zero case we need lemma 3.2.1. □

Theorem 4.3.4 (Syntactic validity)**.** *If* $\Gamma \vdash A = B$ *then* $\Gamma \vdash A$ *and* $\Gamma \vdash B$*. Similarly, if* $\Gamma \vdash t = u : A$ *then* $\Gamma \vdash t : A$ *and* $\Gamma \vdash u : A$*. Finally, if* $\Gamma \vdash t : A$ *then* $\Gamma \vdash A$*.*

Proof. By theorem 4.3.1 and lemma 4.2.18. □

Corollary 4.3.5 (Syntactic validity of Π)**.** *If* $\Gamma \vdash \Pi\, F\, G$ *then* $\Gamma \vdash F$ *and* $\Gamma, F \vdash G$*.*

Proof. By theorems 4.3.2 and 4.3.4. □

Theorem 4.3.6 (Single substitution)**.** *Given* $\Gamma \vdash t : F$ *and either* $\Gamma, F \vdash G$ *or* $\Gamma \vdash \Pi\, F\, G$ *then* $\Gamma \vdash G[t]$*.*

Proof. By theorem 4.3.1, lemma 4.2.10 and lemma 4.2.18. For $\Pi\, F\, G$ we need 4.3.5. □

Theorem 4.3.7 (Inversion)**.** *Given either* $\Gamma \vdash \mathsf{zero} : C$ *or* $\Gamma \vdash \mathsf{suc}\, n : C$ *then* $\Gamma \vdash C = \mathbb{N}$*. Similarly, if* $\Gamma \vdash \mathsf{natrec}\, F\, z\, s\, n : C$ *then* $\Gamma \vdash C = F[n]$*. Finally, if* $\Gamma \vdash f\, a : C$ *then there exists* F *and* G *such that* $\Gamma \vdash f : \Pi\, F\, G$ *and* $\Gamma \vdash a : F$ *and* $\Gamma \vdash C = G[a]$*.*

Proof. By induction on the typing derivation. In cases $\mathsf{natrec}\, F\, z\, s\, n$ and $f\, a$ we need theorem 4.3.6. □

[10] The module `Definition.LogicalRelation.Consequences` contains the formalisation of the consequences of the fundamental lemma.

5 Discussion

In this chapter the proof and its implications will be discussed.

5.1 Implications of proof

Here we will present what it means for the fundamental theorem and its consequences to be proven.

With the fundamental theorem proven, it is possible to take any well-formed construct and show that it is valid. This means that as long as the judgemental rules are followed, the properties of the logical relation for the corresponding are satisfied.

With the well-formedness lemma together with logical relation projection, any valid construct is also well-formed. As such we can move between well-formed, sound and valid, meaning that any proof that can be done in one setting can be done in the others.

Also, the consequences of fundamental theorem follow, meaning that Π injectivity and canonicity hold (theorems 4.3.2 and 4.3.3). Single term substitution for types also follows, meaning that it is safe to substitute any variable, given it is in the context. It is also possible to derive the types of natural numbers, natrec recursion and applications.

Something that is rarely brought up, but done in this thesis is the use of explicit derivations of a logical relation defined with induction-recursion as well as the proof irrelevance of these derivations. Usually, the proof irrelevance is assumed, and with the formalisation we can be sure that it holds. As such, one uncertainty is eliminated.

5.2 Challenges

In this section some of the challenges of formalising the proof are presented. Different solutions are here discussed and the final solution to certain problems are motivated.

5.2.1 Grammar

It is very important that the grammar is formed in a good way, as issues can later on arise. These can make the proof harder to formalise, or even make the proof impossible to prove.

One of the questions was how natrec was to be formalised in the grammar. Initially, it was formed as $\mathsf{natrec}\,(\lambda\,F)\,z\,s$, a function of type $\Pi\,\mathbb{N}\,F$ such that an $n : \mathbb{N}$ has to be applied, and the binder is here instead a function. To simplify the construct and its reduction, the λ and application are embedded in the expression.

An other possible simplification of $\mathsf{natrec}\,F\,z\,s\,n$ is to make s a binder similar to F. Then $\Pi\,\mathbb{N}\,(F \to F[\mathsf{suc}\,\mathsf{x}_0]{\uparrow})$ would be replaced with $\Gamma, \mathbb{N}, F \vdash s : F[\mathsf{suc}\,x_1]{\uparrow}$ instead of $\Pi\,\mathbb{N}\,(F \to F[\mathsf{suc}\,\mathsf{x}_0]{\uparrow})$. This could simplify the lemmas as it would no longer be a function but rather depend on its context.

A consideration for the grammar was to index the terms, such that its type can tell the scope of the variables. However this created complications when implementing the judgements, as each term has then to match the size of the context. It would also make it tricky to implement the well-formed context, as each term in the context can depend on the ones before it.

A consideration was to use a base context instead of an id context such that $\mathsf{base} : \epsilon \subseteq \epsilon$. Then id would be equal to $\mathsf{lift}^n\,\mathsf{base}$ where n is an arbitrary number. This would make it so that there were a unique weakening for each weakening from a fixed context to an other. However, for this to function well, it would require that the terms were indexed, as otherwise it would not be possible to get the correct identity weakening for a certain pair of contexts. Thus, id was used instead as presented in the grammar.

There are different possible representations for substitutions. The formalisation represent them as functions, mapping variable indexes to terms, which proved to be a simple and efficient solution. Before this a list representation was used, however this has certain limitations. For instance, it would not be possible to weaken the substitution list unless it contained all variables of a term, which is only possible to know with indexed terms.

5.2.2 Logical relation

For the lemmas of validity judgements concerning Π-types and λ-abstractions and natrec, there were a couple of challenges that appeared. This was mainly due to the sheer size of the lemmas. All of these lemmas depended on a multitude of the lemmas before them. This made it tricky to figure out which lemmas where needed and if a lemma was missing.

	Lines of code	**Time spent**
Grammar	270	3 weeks
Grammar lemmas	333	
Judgements	253	4 weeks
Judgements lemmas	601	
Logical relation	291	6 weeks
LR. lemmas	1304	
Validity judgements	74	11 weeks
V.J. lemmas	2731	
Fundamental theorem	515	
Consequences	229	
Total	6601	24 weeks

Table 5.1: Table of the number of lines of code and time spent on the different parts of the formalisation.

5.3 Formalisation size and time

In this section we present the size of the formalisation and the time it took to implement. The formalisation was mostly written by one person, with an estimation of 30 hours per week. The times for each section are not precise since a lot of the different parts were written in parallel. In table 5.1 the number of lines of code and time spent on the different sections are presented.

The grammar of the language with weakening and substitution as well as their lemmas consists of 603 lines of code, the largest part being lemmas for propositional equality of the grammar with weakening and substitution. The grammar definition with weakening and substitution was fully realised in 3 weeks. The biggest challenge here which took time was substitution representation and implementation. The propositional equalities for the grammar does not have a clear completion time as the other parts all depends on these proofs with new additions.

For the judgements, with the definitions consisting of 253 lines and the lemmas consisting of 601 lines of code. It took 4 weeks from the grammar definition to fully realise the judgements and their lemmas. What took most time here was the weakening lemma (3.2.5).

The logical relation definition has 291 lines of code and its lemmas consists of 1304 lines of code. From the judgements' completion the logical relation and most of its lemmas took 6 weeks to complete. What here took time was a combination of weakening (4.1.6), equality view (4.1.1), proof irrelevance (4.1.4) and type conversation (4.1.7), where the Π-cases of these lemmas are the most complex. For the size, the number of lines are fairly evenly distributed between the lemmas, with the Π-cases of these lemmas using the most lines.

The validity judgements definition has 74 lines of code and its lemmas consists of 2731 lines of code, the `natrec` lemma being the biggest part consisting of 971 lines of code. After the completion of the logical relation, it took 11 weeks to complete the validity judgements and at the same time completed the fundamental theorem. What required the most time here and the most complex where the Π validity (4.2.12), λ validity (4.2.14) and natural recursion validity (4.2.17).

Finally, the fundamental theorem (4.3.1) consist of 515 lines of code and its consequences consists of 229 lines of code. This was written highly in parallel with the validity judgements lemmas, as the fundamental theorem depend on them, and as such the it was completed at the same time.

All in all, the full formalisation consists of 6601 lines of code. The total time to complete the formalisation was 24 weeks.

5.4 Future works

In this section we present possible future extensions or applications of formalisation.

5.4.1 More types

The formulation of this thesis can also be extended with further types. What follows here are a couple suggestions, some of which are from the full MLTT definition.

One of the most prominent types is the dependent sum type Σ. This will most likely be equally complex as the Π-type formulations, considering they both are dependent types. Σ-types can also function as non-dependent product types[1] if the dependency is not used.

Some other types are finite types, examples being the empty, unit and boolean type and disjunct sum types. It should be easy to extend the formalisation with these, however note that the number of types will be increased, meaning that the logical relation will be more complex. This also means that every lemma and theorem dealing with type and term constructs has to be extended.

An other type that can be interesting to extend the formalisation with is the propositional equality type.

Finally, a tricky but interesting extension is declaration of arbitrary inductive types. An idea would be to extend the context definition with inductive type definitions and implement a primitive recursion method for each inductive type.

5.4.2 Decidability of conversion

One of the more interesting uses of the formalisation is prove the decidability of conversion. This can be accomplished by further formalising Abel and Scherer's

[1]Not to be confused with dependent product types which refer to Π-types.

proof which this thesis is based on [4]. Originally, this thesis was planned to cover this extension as well, however it is left as a future work due to time constraints.

5.4.3 Conversion checking in cubical type theory

An application of the formalisation once it is extended with decidability of conversion is that it can function as a conversion checker for cubical type theory [11]. While this application is not immediate since there needs to be changes to adapt it to that type theory, this formalisation still provides a base for conversion checking.

5.4.4 Consistency

An oversight when writing the formalisation was to require that terms of a neutral type, also were themselves neutral. If this is revised, it would most likely be possible to prove consistency, by proving that $\mathsf{U} \vdash t : \mathsf{x_0}$, or, written with named variables, $X : \mathsf{U} \vdash t : X$, is impossible. An other way to prove consistency would be to introduce the empty type $\bot$ and show that $\epsilon \vdash t : \bot$ is impossible.

6

Conclusions

It has been shown that a proof of Π-injectivity and canonicity of a MLTT language can be formalised in Agda. This proof should most definitely hold true, given that there is confidence in the validity of an Agda proof. Additionally, none of the experimental features of Agda have been used for the formalisation.

It has also been shown that Agda is capable of formalising a simplification of MLTT. The induction-recursion method is used to construct the relations of types, terms, and their respective equalities, as it would otherwise be rejected by its highly mutual dependencies.

Formalisation of a proof is not easy, even with a clear method outlined. Things that are seen insignificant or trivial in a proof on paper are often a bit more trickier, often due to formalisation requiring almost every detail. However, while this requires more work, the confirmation of a proof and a very strongly typed program are two worthwhile consequences of the formal proof.

Bibliography

[1] J. Chapman, "Type theory should eat itself," *Electronic Notes in Theoretical Computer Science*, vol. 228, pp. 21–36, 2009, proceedings of the International Workshop on Logical Frameworks and Metalanguages: Theory and Practice (LFMTP 2008).

[2] ——, "Type checking and normalization," Ph.D. dissertation, School of Computer Science, University of Nottingham, 2009.

[3] T. Altenkirch and A. Kaposi, "Type theory in type theory using quotient inductive types," in *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, ser. POPL '16. New York, NY, USA: ACM, 2016, pp. 18–29.

[4] A. Abel and G. Scherer, "On Irrelevance and Algorithmic Equality in Predicative Type Theory," *Logical Methods in Computer Science*, vol. 8, no. 1, pp. 1–36, 2012, TYPES'10 special issue.

[5] A. Abel, T. Coquand, and B. Mannaa, "On decidability of conversion in type theory," in *22nd International Conference on Types for Proofs and Programs, TYPES 2016, Novi Sad, Serbia, May 23-26, 2016, Book of Abstracts*, S. Ghilezan and J. Ivetic, Eds. EasyChair, 2016.

[6] B. C. Pierce, *Types and Programming Languages*, 1st ed. The MIT Press, 2002.

[7] P. Martin-Löf, *Intuitionistic Type Theory: Notes by Giovanni Sambin of a series of lectures given in Padova, June 1980*, 1984.

[8] "The Agda Wiki." [Online]. Available: http://wiki.portal.chalmers.se/agda/pmwiki.php

[9] G. Plotkin, *Lambda-definability and logical relations.* Edinburgh University, 1973.

[10] P. Dybjer, "A general formulation of simultaneous inductive-recursive definitions in type theory," *The Journal of Symbolic Logic*, vol. 65, no. 02, pp. 525–549, 2000.

[11] C. Cohen, T. Coquand, S. Huber, and A. Mörtberg, "Cubical type theory: a constructive interpretation of the univalence axiom," 2015.