Type theory in a type theory with quotient inductive types

by

Ambrus Kaposi

Thesis submitted to the University of Nottingham
for the degree of Doctor of Philosophy

2017

Abstract

Type theory (with dependent types) was introduced by Per Martin-Löf with the intention of providing a foundation for constructive mathematics. A part of constructive mathematics is type theory itself, hence we should be able to say what type theory is using the formal language of type theory. In addition, metatheoretic properties of type theory such as normalisation should be provable in type theory.

The usual way of defining type theory formally is by starting with an inductive definition of precontexts, pretypes and preterms and as a second step defining a ternary typing relation over these three components. Well-typed terms are those preterms for which there exists a precontext and pretype such that the relation holds. However, if we use the rich metalanguage of type theory to talk about type theory, we can define well-typed terms directly as an inductive family indexed over contexts and types. We believe that this latter approach is closer to the spirit of type theory where objects come intrinsically with their types.

Internalising a type theory with dependent types is challenging because of the mutual definitions of types, terms, substitution of terms and the conversion relation. We use induction induction to express this mutual dependency. Furthermore, to reduce the type-theoretic boilerplate needed for reasoning in the syntax, we encode the conversion relation as the equality type of the syntax. We use equality constructors thus we define the syntax as a quotient inductive type (a special case of higher inductive types from homotopy type theory). We define the syntax of a basic type theory with dependent function space, a base type and a family over the base type as a quotient inductive inductive type.

The definition of the syntax comes with a notion of model and an eliminator: whenever one is able to define a model, the eliminator provides a function from the syntax to the model.

We show that this method of representing type theory is practically feasible by defining a number of models: the standard model, the logical predicate interpretation for parametricity (as a syntactic translation) and the proof-relevant presheaf logical predicate interpretation. By extending the latter with a quote function back into the syntax, we prove normalisation for type theory. This can be seen as a proof of normalisation by evaluation.

Internalising the syntax of type theory is not only of theoretical interest. It opens the possibility of type-theoretic metaprogramming in a type-safe way. This could be used for generic programming in type theory and to implement extensions of type theory which are justified by models such as guarded type theory or homotopy type theory.

Acknowledgements

First of all, I would like to thank my supervisor, Graham Hutton for first accepting me as his PhD student and then giving me the freedom to work on anything I was interested in. He was always available when I needed help and never forgot to give positive feedback on whatever I was doing. I feel very lucky that he was my supervisor even if we didn't end up working together.

I would also like to thank Thorsten Altenkirch for teaching me type theory, suggesting inspiring research topics and spending vast amounts of time with me at the whiteboard. I am also grateful to him for helping me become a member of the type theory research community.

Many thanks to Venanzio Capretta for making valuable suggestions during my annual reviews and for agreeing to be the internal examiner for my thesis defense.

I am grateful to Neil Ghani for being my external examiner and giving helpful feedback. Venanzio and Neil spent a lot of time with my thesis and gave suggestions that improved the presentation significantly.

The Functional Programming Lab in Nottingham was a very pleasant environment to study in. It was relaxed, inspiring and social all at once. I will miss the FP lunches, Friday afternoon pub sessions, Kayal dinners, Away Days and occasional hikes in the Peak District. Special thanks to Nicolai Kraus, Florent Balestrieri, Ivan Perez Dominguez and Nuo Li for their shared interest in sports.

Many thanks to my fellow PhD students Nicolai Kraus, Paolo Capriotti, Florent Balestrieri, Christian Sattler, Nuo Li, Ivan Perez Dominguez and Gabe Dijkstra who taught me a lot about type theory and functional programming and were always available to answer my questions or explore a problem together. I would also like to thank the other members of the FP lab for their contributions to our research environment: Manuel Bärenz, Patrick Bahr, Jan Bracker, Joey Capper, Laurence Day, Jon Fowler, Bas van Gijzel, Jennifer Hackett, Henrik

Nilsson, Jakob von Raumer, Juan Carlos Saenz Carrasco, Neil Sculthorpe.

I would also like to thank the following people outside Nottingham for inspirational discussions on topics related to type theory: Andreas Abel, Matteo Acerbi, Steve Awodey, Flavien Breuvart, Guillaume Brunerie, Thierry Coquand, Zoltán Csörnyei, Péter Diviánszky, Fredrik Nordvall Forsberg, Gergely Éliás, Csaba Hruska, Simon Huber, Tamás Kozsik, Marc Lasson, Andres Löh, Bassel Mannaa, Conor McBride, Guilhem Moulin, Anders Mörtberg, Federico Orsanigo, Gábor Páli, Andrew Polonsky, Marcelo Sousa, Wouter Swierstra, Andrea Vezzosi and the many others whom I forgot to mention.

Outside the FP lab, I would like to thank the Nottingham people Thomas Edwards, Jakub Köry, Francesca Craparotta, Anton De Spiegeleer, the members of the Nottingham Catholic Society and all the other mates in Beeston and Nottingham for their friendship and support.

Finally, I would like to thank my family and friends from home who were missing me when I was in Nottingham and I would like to express my special gratitude to Laura Le Brocq.

Contents

Chapter 1

Introduction

Type theory [58] was introduced by Per Martin-Löf with the intention of providing a foundation for constructive mathematics [69]. A part of constructive mathematics is type theory itself, hence we should be able to say what type theory is using the formal language of type theory. In addition, metatheoretic properties of type theory such as normalisation should be provable in type theory.[1]

One important difference between type theory and set theory is that in type theory every object comes with its type. In set theory everything is given as a set, e.g. natural numbers are just sets and we can construct the disjoint union of two natural numbers, which no one would do because of the intuitive understanding of types. In type theory this intuitive understanding is explicit and such nonsensical constructions are not possible. As Reynolds [89] says:

> "Type structure is a syntactic discipline for enforcing levels of abstraction."

When defining type theory inside type theory, we would like to follow this spirit and define the syntax of type theory in a typed way. The main judgement in type theory expresses that a term t has type A in context Γ:

$$\Gamma \vdash t : A$$

We would like to internalise this as an element of a family of types Tm which is

[1] Note that Gödel's incompleteness theorem applies here: normalisation implies consistency, and we can't prove consistency of a theory in itself: the outer type theory needs more strength (in our case, this means that it has more universes).

indexed by the context and the type:

$$t : \mathsf{Tm}\,\Gamma\,A$$

This way we only define well typed terms. This is in contrast with the usual way of first defining a set of preterms, and then separately defining the ternary typing relation $- \vdash - : -$ over contexts, preterms and types.

In this thesis, we pursue the typed approach and define the syntax of a basic type theory in type theory, define some models including the logical predicate interpretation for parametricity and prove normalisation using a model construction.

The novelty of our approach is the treatment of the conversion relation: instead of defining conversion as a separate relation, we define the syntax as a quotient inductive type (QIT) with equality constructors for conversion rules. QITs are special cases of higher inductive types which come from homotopy type theory [88]. This makes the usage of the internal syntax feasible in practice.

1.1 Background

Martin-Löf's type theory [58] is a formal language intended as a foundation for constructive mathematics based on the Brouwer-Heyting-Kolmogorov interpretation of logic [98]. Unlike traditional foundations where the logic and the theory are separated, propositions and types (sorts) are identified here. Propositions are types, proofs are elements of the type corresponding to the proposition. Implication and universal quantification are (dependent) function space, conjunction and existential quantification are Σ types.

Type theory includes two forms of equalities. One is the conversion relation (judgemental equality, definitional equality) which is a kind of judgement in type theory. When viewing type theory as a programming language [71], terms represent programs and the conversion relation tells us how to run these programs. We can convert one program into another if they are related by the conversion relation. After possibly many such conversion steps, we reach a special form of the program called normal form which is the result of running the program. This process is called *normalisation*. The other equality is the identity type which is a type former that can appear in propositions (types).

The 1972 and 1973 versions of Martin-Löf's type theory [69, 72] are called intensional theories. In these, the conversion relation is decidable and expresses the obvious equalities: equalities that don't need a proof, such as equality by definition. In intensional type theory, the identity type can be viewed as the kind of equality that needs a proof. The 1984 version of type theory [70] is called extensional. In this theory, the two kinds of equalities are identified by the equality reflection rule, conversion becomes undecidable.

Most programming languages (proof assistants) implementing type theory use the intensional variant because of the decidable conversion relation which makes typechecking decidable. Examples of such languages are Coq [73], Agda [80, 97], Idris [30] and Lean [45]. However there are advantages of extensional type theory: it validates the extensional concepts of functional extensionality and quotient types. In addition, its notation is very concise. The price of the conciseness is the lack of information in the syntax which makes conversion undecidable. NuPRL [41] is a proof assistant based on extensional type theory.

Over the years, models [8, 57] and extensions [19] of intensional type theory have been proposed which justify some of the above mentioned extensional concepts. More recently, a new version of intensional type theory, homotopy type theory (HoTT) has emerged [88] based on an interpretation of type theory in abstract homotopy theory. HoTT is incompatible with the equality reflection rule however it is more extensional than extensional type theory itself (if by extensionality we mean that indistinguishable objects are identical): the univalence axiom in HoTT not only validates functional extensionality but also makes isomorphic types equal. In addition, higher inductive types in HoTT generalise quotient types. Higher inductive types are inductively generated types which may have equality constructors. An example is the interval type: it has two point constructors and an equality constructor which says that the two points are equal. The computational content of the new axioms in HoTT is a topic of active research. A constructive model construction [28] and a syntax for a type theory which validates them [40] has been recently given.

Inductive types such as natural numbers are already present in the first formulation of type theory. A general formulation of inductive types called W-types is given in [70]. This covers most usual inductive definitions such as booleans, lists, binary trees. The more general inductive families have been introduced by [49]. These can express types such as lists indexed by their lengths. Motivated by defining the internal syntax of type theory, inductive inductive types were introduced [79]. These can handle the mutual definition of a type together with a family indexed over this type.

To reach our goal of representing the well-typed syntax of type theory inside type theory, we need some extensional concepts in our metatheory. We will use functional extensionality and a special case of higher inductive types combined with inductive inductive types which we call quotient inductive inductive types.

1.2 Related work

In this section, we first summarize the different approaches to internalisation of type theory. Then we explain how the theory of parametricity was extended to dependent types. Finally, we recall the history of normalisation by evaluation and the work on extending it to dependent types.

1.2.1 Internalising type theory

Internalising the syntax of type theory allows reflection and thus generic programming and reasoning about the metatheory in a convenient way.

One way to do this is to make use of the fact that the metatheory is similar to the object theory, e.g. binders of the object theory can be modelled by binders of the metatheory. This way α-renaming and substitution come from the metatheory for free. This approach is called higher order abstract syntax (HOAS) and can be seen as a form of shallow embedding. A generic framework for representing syntax using HOAS is called Logical Framework (LF) [54]. Examples of implementations of LF are Twelf [83] and Beluga [84]. Palsberg and coauthors define self-representations of System U [31] and System F_ω [32] using HOAS. In the latter work they also define a self-interpreter.

The work of Devriese and Piessens [46] provides a very good motivation for this thesis by presenting a typed framework for dependently typed metaprogramming. It differs from our work by representing typing in an extrinsic way i.e. having a type of preterms and a separate typing relation.

McBride [75] presents a deep embedding where the object theoretic judgemental equality is represented by the meta theoretic equality. Our work differs from this because we would like to interpret equality in an arbitrary way and not be restricted by the equality of the meta theory. Thus, we need a concrete representation of equality in our internal syntax.

The work of Chapman [37] is the closest predecessor to our work. He represents the typed syntax of simple type theory and proves normalisation using a big-step semantics [10]. He extends this method to dependent type theory [38] however he does not prove normalisation. In this latter work he encodes judgemental equality as an inductive relation defined separately for contexts, types, substitutions and terms thereby introducing a large inductive inductive definition. While this is a remarkable achievement, it is hard to work with in practice as it suffers from the boilerplate introduced by the explicit equality relations. One needs to work with setoids and families of setoids instead of types, sometimes called the "setoid hell".

The work of Danielsson [43] is also very close to our work. He represents the typed syntax of a dependent type theory using an inductive recursive type i.e. with implicit substitutions. A normalisation function is also defined however without a soundness proof. The usage of implicit substitution seems to give its definition a rather ad-hoc character.

A nice application of type-theoretic metaprogramming is developed by Jaber and coworkers [60] where a mechanism is presented which safely extends Coq with new principles. This relies on presenting a proof-irrelevant presheaf model and then proving constants in the presheaf interpretation. Our approach is in some sense complementary in that we provide a safe translation from well typed syntax into a model, but also more general because we are not limited to any particular class of models.

Our definition of the internal syntax of Type Theory is very much inspired by categories with families (CwFs) [48, 58]. Indeed, one can summarize our work by saying that we construct an initial CwF using QITs.

The style of the presentation can also be described as a generalized algebraic theory (the notion comes from Cartmell [36]) which has been recently used by Coquand to give very concise presentations of Type Theory [28]. Our work shows that it should be possible to internalize this style of presentation in type theory itself.

Higher inductive types are described in chapter 6 of the book on Homotopy Type Theory [88]. Our metatheory is not homotopy type theory, we only use a special case of higher inductive types called quotient inductive types. However we describe some difficulties which arise when we have univalence in the metatheory at the end of section 3.6.

1.2.2 Parametricity for dependent types

Logical relations were introduced into computer science by Tait [96] for proving strong normalisation of simple type theory. The idea is to define a family of relations by induction on types and showing that every term former respects the relation corresponding to their type. Plotkin, Friedman and Statman used logical relations to characterize the definable terms in simple type theory [51, 86, 87, 93]. The term logical relation was coined by Plotkin.

Reynolds used (binary) logical relations to characterise the idea of abstraction in the context of System F: a definition is abstract if it respects logical relations [89]. He called the fundamental theorem of the logical relation (which says that every term respects its corresponding logical relation) parametricity (or abstraction theorem). He also proved an identity extension lemma which says that if we interpret every type variable using the identity relation, then the logical relation for every type will be the identity relation as well. A simple consequence of parametricity is that a function of type $\Pi(A : \mathsf{U}).A \to A$ must be the identity function as there is no way to inspect the type A and such a function needs to work for every possible type A uniformly.

Wadler popularised Reynold's idea later by calling instances of the abstraction theorem "theorems for free" [99] in the context of functional programming in languages based on System F. A summary of Reynold's program for parametricity is given in Reddy's paper [55].

Reynolds presents parametricity using a model construction where the model satisfies the parametricity theorem. Plotkin and Abadi [85] presents a logic in which one can reason about parametricity of System F terms. Bernardy at al [25] observe that the language of type theory is expressive enough to express its own parametricity theorems, hence they define logical relations as a syntactic translation from a pure type system to itself. However they don't give a counterpart of the identity extension lemma. Atkey et al [21] give a presheaf model of dependent type theory which satisfies parametricity and the identity extension lemma. They also show that the existence of (indexed) inductive types follows from parametricity and the identity extension lemma. Keller and Lasson [62] apply Bernardy's syntactic translation to the calculus of constructions and show how to extend the Coq proof assistant with automatic derivation of parametricity theorems.

Bernardy and Moulin defined a type theory with internal parametricity [26, 27, 77]. In these theories, it is possible to prove internally that e.g. the only function of type $\Pi(A : \mathsf{U}).A \to A$ is the identity. It seems that internalising parametricity requires syntax for higher dimensional cubical relations. These arise as a binary relation can be viewed as a type of lines, iterating the parametricity construction on a binary relation creates a type of squares which can be seen as a two-dimensional relation, one more iteration gives types of 3-dimensional cubes etc. This makes these theories fairly complex systems. A simplification inspired by cubical type theory [40] can be achieved by introducing an interval pretype into the syntax [24, 78]. This theory also has a presheaf model showing the connection to the models of Atkey [21] and Coquand [28].

In chapter 4, we show how to formalise the logical predicate interpretation of dependent types [25] using our presentation of the syntax of type theory, internally.

In section 4.3 we use logical predicates to derive the eliminator for a closed quotient inductive inductive type (a combination of higher inductive types and inductive inductive types, see section 2.4). Related to this is the work of Ghani et al on generic fibrational induction [52]. Their work is semantic in nature and they derive the induction rule for an inductive type defined by a functor. They don't derive the existence of inductive types from parametricity as done e.g. by Atkey [21]. They don't mention logical relations explicitly however they use similar ideas in a categorical setting, e.g. a morphism between predicates is a function which preserves the predicates. Our work is incomplete in that we don't prove correctness of our approach and we only treat closed types however it scales to inductive inductive and higher inductive types. Another difference is that our method is given as a syntactic translation.

1.2.3 Normalisation by evaluation

Traditional normalisation proofs work by showing that the small step reduction relation is confluent and strongly normalising. An example is [53] where this approach is applied to simple type theory, System F and System T. [7] is using similar methods to prove decidability of conversion for a dependent type theory.

Martin-Löf's first formulation of type theory [69] includes a proof of normalisation for well-typed terms using a semantic argument in a constructive metalanguage. This was the first example of normalisation by evaluation (NBE) for type theory.

Independently, Berger and Schwichtenberg [23] implemented a normalisation algorithm for simple type theory which uses evaluation in the host language (Scheme) to normalise terms in the object language; they define a quote function which turns an evaluated program into a normal form. They coined the term normalisation by evaluation.

A categorical reconstruction of the proof [23] was given in [13]. Here well-typed terms are evaluated in a presheaf model and correctness of normalisation is proved using a logical relation (which can be viewed as a variant of categorical glueing). The same authors extended their proof to System F [14, 15] and to strong coproducts [11]. Our work can be seen as a continuation of this line of research.

The term normalisation by evaluation is also more generally used to describe semantic based normalisation functions. E.g. Danvy is using semantic normalisation for partial evaluation [44]. Normalisation by evaluation using untyped realizers has been applied to dependent types by Abel et al [4–6].

Danielsson [43] formalized big-step normalisation for dependent types without a proof of soundness or completeness.

1.3 How to read this thesis

The thesis is structured in the following way.

Chapter 2 introduces the metatheory and notation that we use throughout this thesis. The notation we use is very close to that of the proof assistant Agda [80, 97]. We work in intensional type theory with quotient inductive inductive types (QIITs), a strict equality type (that is, with uniqueness of identity proofs, UIP) and functional extensionality. Although we work in an intensional type theory (and the formalisation is performed in this theory), for ease of notation, we use extensional type theory to write down equational reasoning. Knowing that proofs in extensional type theory can be translated to intensional type theory with UIP and functional extensionality [56,81], we can view this notational convenience as acceptable. We show the difference between the two kinds of reasoning through a few examples in section 3.3.

In chapter 3, we define the internal syntax of type theory and define the standard model. We start by introducing an informal syntax for type theory and show what steps we need to take to arrive at a formal syntax. After the formal definition of the syntax, we show how to reason with it e.g. we show how to derive the substitution laws for terms and the polymorphic identity function. We show how to define functions from the syntax using the eliminator and also give some examples. We note that the arguments of the eliminator constitute a well-known notion of model of type theory called categories with families [48]. An example of such a model that we give is the standard model (metacircular interpretation) where every object theoretic construct is interpreted by its metatheoretic counterpart.

In chapter 4 we show how to define logical predicates and relations for the informal type theory given in the previous chapter, and then we show how to define the logical predicate interpretation for our formal theory using the eliminator of the syntax. This interpretation is a syntactic translation, it targets the syntax itself. As a practical usage of this interpretation, we sketch an algorithm for deriving the type of the eliminator for a closed QIIT defined as a context. We used this algorithm to derive the eliminator for our definition of the syntax.

Chapter 5 proves normalisation for our type theory. Normalisation is specified as an isomorphism between terms and normal forms, the correctness of normalisation (soundness and completeness) follows from this. We prove normalisation using the technique of normalisation by evaluation which involves a model construction and then a quote function from the model to normal forms. The model that we use is a proof-relevant presheaf logical predicate. We show how consistency follows from normalisation.

Chapters 4 and 5 don't depend on each other and can be read independently. In chapter 4, we use "arguments of the eliminator" notation for defining a function from the syntax, while in chapter 5 we use recursive notation (except section 5.4 on the presheaf model which uses "arguments of the eliminator" notation as well).

Chapter 6 summarizes the thesis and sketches future directions of research.

In this thesis, we make use of three different type theories: the metatheory, an informal and a formal object theory. We use different notations to distinguish between them.

- Our metatheory (chapter 2) is close to the notation of Agda, e.g. we write $(x : A) \to B$ for a function type.

- We use an informal object theory (section 3.1, figure 3.1) for the illustration of the differences between the informal and formal theories. Function types are written as $\Pi(x : A).B$.

- Our formal theory (section 3.2) uses De Bruijn variables and explicit substitutions, function types are written as $\Pi\, A\, B$.

It is recommended that the reader of this thesis has a basic understanding of type theory. A nice introduction to the basic concepts is chapter 1 in [88]. For chapter 5, category theoretic intuition is helpful as we use a few categorical concepts for organisation. However, we define all of these rigorously.

1.4 List of contributions and publications

The main contributions of this thesis are the following.

- We define a workable syntax of type theory in type theory using intrinsic typing. We present a basic type theory with Π and an uninterpreted base type and an uninterpreted family over the base type (chapter 3).

- We demonstrate the usability of this internal syntax by proving basic properties: disjointness of context and type constructors (sections 3.4.1) and injectivity of Π (section 3.5.1).

- We define the standard model (section 3.6). In this model, every object theoretic construct is modelled by its metatheoretic counterpart.

- We formalise the internal logical predicate interpretation of type theory (chapter 4). This allows the derivation of parametricity theorems for terms of type theory using the same theory.

- We prove normalisation for our basic type theory using the technique of normalisation by evaluation (chapter 5). Using normalisation, we prove consistency of our theory.

- We describe an algorithm for deriving the type of the eliminator for a closed QIIT specified as a context of type formation rules and constructors (section 4.3).

- Most of the constructions in this thesis were formalised in the proof assistant Agda (see section 1.5).

The material of chapters 3 and 4 first appeared in the paper [18] and the contents of chapter 5 first appeared in [16]. Other publications of the author written during his PhD which are not immediately related to this thesis are [35] and [17].

1.5 Formalisation

Most of the contents of this thesis were formalised using the proof assistant Agda [80,97]. The formalisation is available online [61] and is supplied with a "readme" file describing where to find each part of the thesis.

We list the status of the formalisation for each chapter as of writing this thesis.

- Most examples in chapter 2 are formalised. The codes for different kinds of inductive types and their decoding are formalised.
- Chapter 3 on the syntax and the standard model is fully formalised.
- In chapter 4 on parametricity, section 4.2 is formalised.
- In chapter 5 on NBE.
  - Sections 5.4, 5.6, 5.8, 5.10 and 5.11 are formalised.
  - The computational parts of sections 5.7 and 5.9 are formalised, most of the naturality proofs are omitted.

We discuss the tools that we use for the formalisation (e.g. how we implement quotient inductive inductive types) in chapter 2 in the corresponding sections.

Chapter 2

Metatheory and notation

The metatheory we use in this thesis is intensional Martin-Löf type theory with a hierarchy of universes, Π types, a strict equality type, inductive inductive types, and quotient inductive types added using axioms. We will describe these features and the notation through examples in the following sections.

Formalisation of most parts of this thesis was performed in Agda. Agda is a functional programming language and a proof assistant implementing Martin-Löf type theory [80, 97]. It has a concise notation for writing down terms in type theory and it checks whether they are derivable (type-checking). It supports modularity by allowing definitions (assigning a name to a term) and collecting these definitions into modules which can be parameterised by a telescope (a part of a context). One can also postulate an element of any given type, this is the same as adding an extra parameter to the current module.

We will use a notation throughout this thesis which is close to Agda's.

2.1 Definitions, universes, Π and Σ

The technical content of this thesis can be viewed as a list of definitions in type theory surrounded by explanations. We use the sign $:=$ for definitions and $=$ for judgemental (definitional) equality. We don't use a separate notation for proofs, they are just type-theoretic definitions and we can refer to them by their name. Later definitions can refer to previous ones.

We overload names, e.g. the action on objects and morphisms of a functor will be denoted by the same symbol.

We write $(x : A) \to B$ or $\forall x.B$ for dependent function space, $\lambda x.t$ for abstraction and $f\,u$ for application. Multiple arguments can be collected together, e.g. $(x\,y : A)(z : B) \to C$, $\lambda x\,y\,z\,.\,t$.

We use the following two notations for defining the name d to be $\lambda x.t$ of type $(x : A) \to B$. The left one uses Coq-style notation, the right one uses Agda style.

$$\mathsf{d}\,(x : A) : B := t \qquad\qquad \begin{aligned} &\mathsf{d} : (x : A) \to B \\ &\mathsf{d}\,x := t \end{aligned}$$

We use the symbol $-$ as a placeholder for arguments, e.g. $- + -$ is a notation for $\lambda x\,y.x + y$.

We sometimes omit arguments of functions when they are determined by other arguments or the return type. For example the full type of function composition is $(A\,B\,C : \mathsf{Set}) \to (B \to C) \to (A \to B) \to (A \to C)$, but the first three arguments are determined by the types of the last two arguments. In this case we write $\{A\,B\,C : \mathsf{Set}\} \to (B \to C) \to (A \to B) \to (A \to C)$ or even $(B \to C) \to (A \to B) \to (A \to C)$. We call the arguments given in curly braces *implicit arguments*. When we call this function, we can omit the implicit arguments or specify them in lower index.

When defining two functions mutually, we first declare them by giving their names and types, then give the definitions one after the other. These definitions can be reduced to a single function definition which provides the result of both functions. We give the example of deciding whether a natural number is even or odd.

$$\begin{aligned}
&\mathsf{isEven} : \mathbb{N} && \to \mathsf{Bool} \\
&\mathsf{isOdd} : \mathbb{N} && \to \mathsf{Bool} \\
&\mathsf{isEven}\ \ \mathsf{zero} && := \mathsf{true} \\
&\mathsf{isEven}\ \ (\mathsf{suc}\,n) && := \mathsf{isOdd}\,n \\
&\mathsf{isOdd}\ \ \mathsf{zero} && := \mathsf{false} \\
&\mathsf{isOdd}\ \ (\mathsf{suc}\,n) && := \mathsf{isEven}\,n
\end{aligned}$$

We can rewrite this definition into one that gives the result of both functions as a pair and is not using pattern matching but the recursor (see next section).

$$
\begin{aligned}
&\mathsf{isEvenOdd} : \mathbb{N} \to \mathsf{Bool} \times \mathsf{Bool}\\
&\mathsf{isEvenOdd} := \mathsf{Rec}\mathbb{N}\,(\mathsf{Bool} \times \mathsf{Bool})\,(\mathsf{true}, \mathsf{false})\,(\lambda p.(\mathsf{proj}_2\, p, \mathsf{proj}_1\, p))\\
&\mathsf{isEven}\,(n : \mathbb{N}) : \mathsf{Bool} := \mathsf{proj}_1\,(\mathsf{isEvenOdd}\, n)\\
&\mathsf{isOdd}\ \ (n : \mathbb{N}) : \mathsf{Bool} := \mathsf{proj}_2\,(\mathsf{isEvenOdd}\, n)
\end{aligned}
$$

We denote the universe of types by Set. We will refrain from writing the universe level, but we assume $\mathsf{Set} = \mathsf{Set}_0 : \mathsf{Set}_1 : \mathsf{Set}_2 : ....$ We work in a strict type theory (see section 2.2.2), hence the naming is appropriate in the sense that sets are types with trivial higher equality structure.

We write $\Sigma(x : A).B$ for dependent sum types using proj_1 and proj_2 for the projections.

Following Agda's notation, we use iterated Σ types with named projections called records. To define such a type, we use the record keyword and list the fields of the record afterwards. The field names become projection functions. The following type can be seen as a variation of $\Sigma(x : \mathsf{A}).\mathsf{B}\, x$ with the projection functions $\mathsf{a} = \mathsf{proj}_1$ and $\mathsf{b} = \mathsf{proj}_2$.

```
record R : Set
  a    : A
  b    : B a
```

Given $r : \mathsf{R}$ we have $\mathsf{a}\, r : \mathsf{A}$ and $\mathsf{b}\, r : \mathsf{B}\,(\mathsf{a}\, r)$. The other way of putting a and b into scope given an $r : \mathsf{R}$ is by saying $\mathsf{open}\,\mathsf{R}\, r$.

Sometimes it is convenient to collect definitions which belong together into one collection called a module. Modules can have parameters which make the concise definition of multiple functions with the same parameters easy. For example the following module M has a parameter of the previous record type R and after opening the record, one can use the fields in all the definitions.

```
module M (r : R)
  open R r
  f : A := a
  g : B f := b
```

We write $A + B$ for the sum type of A and B and we denote the injections by inj_1 and inj_2 and eliminator by $[f, g] : A + B \to C$ for $f : A \to C$ and $g : B \to C$.

We denote the empty type by $\bot$ (with eliminator $\mathsf{Elim}\bot : \bot \to A$ for any A) and the type only having a single constructor tt by $\top$.

2.2 Inductive types

Our metatheory has inductive types: this means that we have an open universe in which any number of inductive types can be defined.

Elements of inductive types are freely generated by their constructors and the only elements belonging to such a type are those created by constructors. This is expressed by the fact that an inductive type comes with an eliminator which lets one define a function from the inductive type by only providing a case for each constructor.

An inductive type is defined by first declaring its sort and then listing the constructors and their types. The constructors need to obey the condition of strict positivity: recursive occurrences can only appear in strictly positive positions [3]. A simple example of an inductive type is natural numbers with sort Set and constructors zero and suc. We use the data keyword to indicate inductive definitions.

$$\begin{aligned}
&\mathsf{data}\,\mathbb{N} : \mathsf{Set}\\
&\quad \mathsf{zero} : \mathbb{N}\\
&\quad \mathsf{suc}\ \ : \mathbb{N} \to \mathbb{N}
\end{aligned}$$

We can define a function from $\mathbb{N}$ to another type using the eliminator of $\mathbb{N}$. The structure of the eliminator is determined by the constructors. There are two kinds of eliminators for an inductive type: the non dependent one (also called the recursor) lets us define a function of type $\mathbb{N} \to A$ for some type A. The dependent eliminator lets us define a dependent function of type $(n : \mathbb{N}) \to A\,n$ where $A : \mathbb{N} \to \mathsf{Set}$. Note that the recursor is a special case of the eliminator. The type of both are given below. We give names to the arguments of the eliminator in a systematic way: we add an ^{M} index to the names of the corresponding type and constructors.

$$
\begin{aligned}
\mathsf{Rec}\mathbb{N} \;&: (\mathbb{N}^M : \mathsf{Set}) \\
&\quad (\mathit{zero}^M : \mathbb{N}^M)(\mathit{suc}^M : \mathbb{N}^M \to \mathbb{N}^M) \\
&\quad \to \mathbb{N} \to \mathbb{N}^M \\
\mathsf{Elim}\mathbb{N} : &\; (\mathbb{N}^M : \mathbb{N} \to \mathsf{Set}) \\
&\quad (\mathit{zero}^M : \mathbb{N}^M\,\mathsf{zero})(\mathit{suc}^M : (n : \mathbb{N}) \to \mathbb{N}^M\,n \to \mathbb{N}^M\,(\mathsf{suc}\,n)) \\
&\quad (n : \mathbb{N}) \to \mathbb{N}^M\,n
\end{aligned}
$$

Following the nomenclature of [76], $\mathbb{N}^M$ is called the *motive* of the eliminator. In the dependent case it can be viewed as a predicate on $\mathbb{N}$: with this view in mind the eliminator coincides with the usual induction principle for natural numbers. zero^M and suc^M are called the *methods* of the eliminator, there is one method corresponding to each constructor. The natural number that we eliminate is called the *target*. The motive and the methods of the recursor together form an *algebra* of the corresponding signature functor [29].

Moreover, we have the following equalities which express that when applying the eliminator on a constructor as a target we use the corresponding method.

$$
\begin{aligned}
\mathsf{Rec}\mathbb{N}\;\; \mathbb{N}^M\,\mathit{zero}^M\,\mathit{suc}^M\,\mathsf{zero} \quad &= \mathit{zero}^M \\
\mathsf{Rec}\mathbb{N}\;\; \mathbb{N}^M\,\mathit{zero}^M\,\mathit{suc}^M\,(\mathsf{suc}\,n) &= \mathit{suc}^M\,(\mathsf{Rec}\mathbb{N}\,\mathbb{N}^M\,\mathit{zero}^M\,\mathit{suc}^M\,n) \\
\mathsf{Elim}\mathbb{N}\,\mathbb{N}^M\,\mathit{zero}^M\,\mathit{suc}^M\,\mathsf{zero} \quad &= \mathit{zero}^M \\
\mathsf{Elim}\mathbb{N}\,\mathbb{N}^M\,\mathit{zero}^M\,\mathit{suc}^M\,(\mathsf{suc}\,n) &= \mathit{suc}^M\,n\,(\mathsf{Elim}\mathbb{N}\,\mathbb{N}^M\,\mathit{zero}^M\,\mathit{suc}^M\,n)
\end{aligned}
$$

In practice, defining natural numbers by giving a data definition means that after the definition we can use $\mathbb{N}$, zero, suc, $\mathsf{Rec}\mathbb{N}$, $\mathsf{Elim}\mathbb{N}$ and the computation rules for $\mathsf{Rec}\mathbb{N}$ and $\mathsf{Elim}\mathbb{N}$ hold.

Agda does not directly supply eliminators for inductive types but it provides pattern matching as a generic mechanism to define functions from inductive types. As eliminators only allow the definition of terminating functions, Agda augments pattern matching with termination checking making sure that all the recursive calls are made at structurally smaller arguments. The eliminator can be implemented by pattern matching as written above. In a strict theory like ours, the other direction is true as well: pattern matching can be translated to eliminators [74].

Whenever we use pattern matching in this thesis, we only do it for sake of readability and it can always be translated to eliminators. For example, the pattern matching style definition of addition is

$$\begin{aligned}&\mathsf{zero} \quad + n := n\\ &\mathsf{suc}\, m + n := \mathsf{suc}\,(m + n)\end{aligned}$$

while using the eliminator it is given as

$$- + - := \mathsf{Rec}\mathbb{N}\,(\mathbb{N} \to \mathbb{N})\,(\lambda n.n)\,(\lambda f\, n.\mathsf{suc}\,(f\, n)).$$

We mentioned that strict positivity is a requirement for the constructors of an inductive type. Induction rules for certain non strictly positive types would make the theory inconsistent. We can make this strict posivitity requirement precise by saying that those inductive definitions are allowed which are given by certain *codes*. For simple inductive types like natural numbers, a code is given by a set S and a $P : S \to \mathsf{Set}$ where S and P are built using dependent function space, $\top, \bot, \mathsf{Bool}$ and Σ. We define a decoding function $[\![-]\!]$ which for each code gives the following function.

$$\begin{aligned}&[\![-]\!] : \big(\Sigma(S : \mathsf{Set}).S \to \mathsf{Set}\big) \to \mathsf{Set} \to \mathsf{Set}\\ &[\![(S, P)]\!]\, X := \Sigma(s : S).P\, s \to X\end{aligned}$$

A code (S, P) determines the following inductive type with one constructor.

$$\begin{aligned}&\mathsf{data}\,\mathsf{W}_{S\,P} : \mathsf{Set}\\ &\quad \mathsf{con} : [\![(S, P)]\!]\,\mathsf{W}_{S\,P} \to \mathsf{W}_{S\,P}\end{aligned}$$

S is called the set of shapes (the set of constructors) and $P\, s$ is called the set of positions for the shape s. Positions determine the number of inductive occurrences. S and P together all called a container [2, 3]. Decoding a container gives a strictly positive functor and the initial algebra of such a functor is called a W-type, hence the name $\mathsf{W}_{S\,P}$.

For example, natural numbers can be encoded by $S := \mathsf{Bool}$ (there are two constructors), $P\,\mathsf{false} := \bot$ (the zero constructor doesn't have any inductive arguments) and $P\,\mathsf{true} := \top$ (the suc constructor has one inductive argument). With this definition of natural numbers, the constructors are given by $\mathsf{zero} := \mathsf{con}\,\mathsf{false}\,\mathsf{Elim}\bot$ and $\mathsf{suc}\,n := \mathsf{con}\,\mathsf{true}\,(\lambda\,\_.n)$.[1]

2.2.1 Inductive families

We also use the more general inductive types called inductive families [49] or indexed inductive types. These are families of types indexed by another type. The prototypical example is vectors, i.e. $\mathbb{N}$-indexed lists where the index stores the vector's length.

$$\begin{aligned}
&\mathsf{data}\,\mathsf{Vec}\,(A : \mathsf{Set}) : \mathbb{N} \to \mathsf{Set}\\
&\quad [] \qquad : \mathsf{Vec}\,A\,\mathsf{zero}\\
&\quad - :: - : A \to \{n : \mathbb{N}\} \to \mathsf{Vec}\,A\,n \to \mathsf{Vec}\,A\,(\mathsf{suc}\,n)
\end{aligned}$$

Here the type A is a parameter of the type (this can be viewed as a variable in the context for the type) while $\mathbb{N}$ is an *index* of the type: different constructors can construct elements of the type at different indices. The $[]$ constuctors creates vector of length zero, while the $- :: -$ constructor takes a vector of length n and turns it into a vector of length $\mathsf{suc}\,n$. The type of the eliminator for Vec is the following.

$$\begin{aligned}
\mathsf{ElimVec} : \;& (A : \mathsf{Set})\big(\mathsf{Vec}^M : (n : \mathbb{N}) \to \mathsf{Vec}\,A\,n \to \mathsf{Set}\big)\\
& \big([]^M : \mathsf{Vec}^M\,\mathsf{zero}\,[]\big)\\
& \big({-}{::}^M{-} : (x : A)\{n : \mathbb{N}\}\{xs : \mathsf{Vec}\,A\,n\} \to \mathsf{Vec}^M\,n\,xs\\
& \qquad\quad \to \mathsf{Vec}^M\,(\mathsf{suc}\,n)\,(x :: xs)\big)\\
& \{n : \mathbb{N}\}(xs : \mathsf{Vec}\,A\,n) \to \mathsf{Vec}^M\,xs
\end{aligned}$$

As in the case of inductive types, we can specify inductive families using codes. Given an indexing set I, the codes are given by a set of shapes and a set of positions for each shape together with functions providing the output and input indices. We collect these into the record C_I.

[1] We remark that functional extensionality (see section 2.2.2) is needed to define the induction principle using the induction principle of $\mathsf{W}_{S\,P}$ (see the discussion in section 2.1 of [19]).

$$
\begin{array}{l}
\mathsf{record}\ \mathsf{C}_I : \mathsf{Set}_1 \\
\quad \begin{array}{lll}
\mathsf{S} & : & \mathsf{Set} \\
\mathsf{P} & : & \mathsf{S} \to \mathsf{Set} \\
\mathsf{out} & : & \mathsf{S} \to I \\
\mathsf{in} & : & (s : \mathsf{S}) \to \mathsf{P}\, s \to I
\end{array}
\end{array}
$$

The output index is what the constructor of a given shape outputs, while the input index is the index of an inductive argument occurring at a given position. We have two decoding functions, one which calculates the type of arguments for the constructor and one which gives the index of an argument.

$$
\begin{array}{l}
[\![-]\!] : \mathsf{C}_I \to (I \to \mathsf{Set}) \to \mathsf{Set} \\
[\![c]\!]\, X := \Sigma(s : \mathsf{S}\, c).\mathsf{P}\, c\, s \to X\, (\mathsf{in}\, c\, s\, p) \\
\mathsf{index} : [\![c]\!]\, X \to I \\
\mathsf{index}\, \{c\}\, (s, \_) := \mathsf{out}\, c\, s
\end{array}
$$

Using these an indexed W-type is given by the following definition.

$$
\begin{array}{l}
\mathsf{data}\ \mathsf{W}_{(I:\mathsf{Set})\,(c:\mathsf{C}_I)} : I \to \mathsf{Set} \\
\quad \mathsf{con} : (w : [\![c]\!]\, \mathsf{W}_{I\,c}) \to \mathsf{W}_{I\,c}\, (\mathsf{index}\, w)
\end{array}
$$

For the type of vectors, the indexing type is $\mathbb{N}$ and the code is given as follows.

$$
\begin{array}{lll}
\mathsf{S} & & := \Sigma(b : \mathsf{Bool}).\mathsf{if}\ b\ \mathsf{then}\ \top\ \mathsf{else}\ A \times \mathbb{N} \\
\mathsf{P} & (\mathsf{true}, \mathsf{tt}) & := \bot \\
\mathsf{P} & (\mathsf{false}, (a, n)) & := \top \\
\mathsf{out} & (\mathsf{true}, \mathsf{tt}) & := \mathsf{zero} \\
\mathsf{out} & (\mathsf{false}, (a, n)) & := \mathsf{suc}\, n \\
\mathsf{in} & (\mathsf{true}, \mathsf{tt})\, c & := \mathsf{Elim}\bot\, c \\
\mathsf{in} & (\mathsf{false}, (a, n))\, \mathsf{tt} & := n
\end{array}
$$

The shape is a boolean and in the case when it is false, an A and a natural number. The true case corresponds to the [] constructor, and the false case to the :: constructor which has an A and an $\mathbb{N}$ non-inductive argument. There are no inductive arguments in the [] case, hence the positions in this case are given by the empty set. In the :: case we have one inductive argument encoded by $\top$. *out* expresses that the index of the [] constructor is zero and the index of the :: constructor is one more than its natural number argument. There are no positions in the [] case, hence we don't need to give any indices for those (we use $\mathsf{Elim}\bot$ to express this) and the index of the single inductive argument of :: is n.

The above codes for inductive families are called indexed containers [12]. Here we presented a slightly different, but equivalent formulation. Indexed W-types can be translated into ordinary W-types [12].

2.2.2 The identity type

A notable example of indexed types is the identity type which we present in the Paulin-Mohring formulation [82]. It is also called propositional equality to distinguish from the conversion relation which is called judgemental or definitional equality.

$$
\begin{aligned}
&\mathsf{data} - \equiv - \{A : \mathsf{Set}\}(a : A) : A \to \mathsf{Set}\\
&\quad \mathsf{refl} : a \equiv a
\end{aligned}
$$

It has two parameters, the implicit parameter $A : \mathsf{Set}$ and an element a of A and it has one index of type A. If this index is b, it expresses that a and b are equal. The single constructor is called reflexivity saying that a is equal to itself. The eliminator is called J and is given as follows.

$$
\begin{aligned}
&\mathsf{J} : \{A : \mathsf{Set}\}\{a : A\}\\
&\quad (Q\ : (x : A) \to a \equiv x \to \mathsf{Set})\\
&\quad (r : Q\,a\,\mathsf{refl})\\
&\quad (x : A)(q : a \equiv x) \to\ Q\,x\,q
\end{aligned}
$$

It expresses that if we have an object of a type and a family over equalities from that object then it is enough to give an element of the family at refl, and we get an element of the family at every other equality as well.

Although this elimination rule seems to say that if we want to prove something depending on an equality, we only need to prove it for refl, we cannot prove that every inhabitant of the equality type $a \equiv a$ is refl [59]. For this, we need the additional axiom K [95] which says the following.

$$\mathsf{K} : (X : \mathsf{Set})(x : X)(p : x \equiv x) \to p \equiv \mathsf{refl}$$

The default way of Agda's pattern matching mechanism satisfies this axiom in addition to J. We will also assume this axiom throughout the thesis.

Using J, we can prove the following important properties of equality.

$$\begin{aligned}
&-^{-1} && : a \equiv b \to b \equiv a \\
&- \centerdot - && : a \equiv b \to b \equiv c \to a \equiv c \\
&\mathsf{coe} && : (A \equiv B) \to A \to B \\
&\mathsf{ap} && : (f : A \to B) \to a \equiv a' \to f\,a \equiv f\,a' \\
&{}_{-*}- && : a \equiv a' \to P\,a \to P\,a'
\end{aligned}$$

Symmetry ($-^{-1}$) and transitivity ($- \centerdot -$) together with the constructor refl make equality of any type an equivalence relation. We can coerce between equal types and we denote this by coe. Equality is a congruence which is expressed by ap ("apply on path"). This means that every function f respects equality.

We will use standard lemmas about ap and $\centerdot$ e.g.

$$\begin{aligned}
&\mathsf{apap} && : \mathsf{ap}\,g\,(\mathsf{ap}\,f\,p) \equiv \mathsf{ap}\,(g \circ f)\,p \\
&\mathsf{ap}\centerdot && : \mathsf{ap}\,f\,(p \centerdot q) \equiv \mathsf{ap}\,f\,p \centerdot \mathsf{ap}\,f\,q
\end{aligned}$$

Transport (substitutivity of equality) is written ${}_{-*}-$. If we have an element u of a type family P at some a, and we know that a equals a' then we can transport u along the equality to get an element of the type family at a'. If the equality is denoted p, then we write the transported element ${}_{p*}u$. Transport can be proved combining coe and ap.

Sometimes we want to express the equality between elements of different types knowing that the types are equal. In this case given $u : P\,a$, $u' : P\,a'$ and a proof $q : a \equiv a'$, we write $u \equiv^q u'$ to abbreviate ${}_{q*}u \equiv u'$.

We will use equational reasoning which is just repeated usage of transitivity and ap. The following example is just a fancy way of writing $r : \big(f\,(f\,a) \equiv f\,c\big) :=$

$\mathsf{ap}\,(f \circ g)\,p \centerdot \mathsf{ap}\,f\,q$ where $p : a \equiv b$ and $q : g\,b \equiv c$.

$$\begin{aligned}
r \ : \ & f\,(g\,a) \\
& \qquad\qquad (p) \\
\equiv\, & f\,(g\,b) \\
& \qquad\qquad (q) \\
\equiv\, & f\,c
\end{aligned}$$

Writing down the transports all the time can be cumbersome and makes the notation hard to read. This is the burden one has to take when working in intensional type theory. In extensional type theory, one can replace an object with a propositionally equal one using the equality reflection rule which makes the notation lighter. E.g. knowing that $A \equiv A'$ and $a' : A'$ then $f\,a'$ is well typed even if f has type $A \to B$. In this thesis, for readability, we omit writing down the transports (with the exception of section 3.3 where we give examples in both notation). That is, we will work informally in extensional type theory. However, we know that everything that we prove this way can be justified in intensional type theory extended with the axioms K and functional extensionality [56,81]. In the Agda formalisation we work in this latter theory.

With the type of codes given in section 2.2.1, the identity type is defined as follows. The indexing type is $A \times A$.

$$\begin{aligned}
&\mathsf{S} && := A \\
&\mathsf{P} \quad a && := \bot \\
&\mathsf{out}\,a && := (a, a) \\
&\mathsf{in} \quad a\,c && := \mathsf{Elim}\bot\,c
\end{aligned}$$

2.2.3 Induction recursion

Induction recursion is a further generalisation of inductive types. We describe it not only because we make use of it but also because it is a good warm-up for comprehending the codes for induction induction.

Induction recursion allows the definition of an inductive type simultaneously with a function defined by recursion over that type. An example is an internal universe of codes for types V and a decoding function $\mathsf{Elem} : \mathsf{V} \to \mathsf{Set}$.

$$\begin{array}{ll}
\mathsf{data}\,\mathsf{V} : \mathsf{Set} & \mathsf{Elem} \;: \mathsf{V} \to \mathsf{Set} \\
\quad \mathsf{zero} : \mathsf{V} & \mathsf{Elem}\,\mathsf{zero} := \bot \\
\quad \mathsf{pi} \quad : (a : \mathsf{V}) \to (\mathsf{Elem}\, a \to \mathsf{V}) \to \mathsf{V} & \mathsf{Elem}\,(\mathsf{pi}\, a\, b) := (x : \mathsf{Elem}\, a) \to \mathsf{Elem}\,(b\, x)
\end{array}$$

Note that the constructor pi refers to the function Elem which is defined by recursion on the type being defined.

The inductive recursive definitions which make sense can be given by a type of codes together with a decoding function. The type of codes is the following.

$$\begin{array}{ll}
\mathsf{data}\,\mathsf{IR}_D & : \mathsf{Set}_1 \\
\quad \mathsf{nil} & : D \to \mathsf{IR}_D \\
\quad \mathsf{nonind} & : (A : \mathsf{Set}) \to (A \to \mathsf{IR}_D) \to \mathsf{IR}_D \\
\quad \mathsf{ind} & : (A : \mathsf{Set}) \to \big((A \to D) \to \mathsf{IR}_D\big) \to \mathsf{IR}_D
\end{array}$$

The code $\mathsf{nil}\, d$ represents a trivial constructor and d is the result of the recursive function at this trivial constructor. $\mathsf{nonind}\, A\, f$ represents a non inductive argument of type A and f gives the rest of the arguments. $\mathsf{ind}\, A\, F$ represents inductive arguments $A \to U$ and F gives the result of the recursive function on them.

There are two decoding functions, one for the constructor and one for the function defined by recursion.

$$\begin{array}{l}
[\![-]\!]_{\mathsf{con}} : \mathsf{IR}_D \to (U : \mathsf{Set}) \to (U \to D) \to \mathsf{Set} \\
[\![\mathsf{nil}\, d]\!]_{\mathsf{con}}\, U\, T := \top \\
[\![\mathsf{nonind}\, A\, f]\!]_{\mathsf{con}}\, U\, T := \Sigma(a : A).[\![f\, a]\!]_{\mathsf{con}}\, U\, T \\
[\![\mathsf{ind}\, A\, F]\!]_{\mathsf{con}}\, U\, T := \Sigma(g : A \to U).[\![F\,(T \circ g)]\!]_{\mathsf{con}}\, U\, T
\end{array}$$

$$\begin{array}{l}
[\![-]\!]_{\mathsf{fun}} : (\gamma : \mathsf{IR}_D)(U : \mathsf{Set})(T : U \to D) \to [\![\gamma]\!]_{\mathsf{con}}\, U\, T \to D \\
[\![\mathsf{nil}\, d]\!]_{\mathsf{fun}}\, U\, T\, \mathsf{tt} := d \\
[\![\mathsf{nonind}\, A\, f]\!]_{\mathsf{fun}}\, U\, T\, (a, x) := [\![f\, a]\!]_{\mathsf{fun}}\, U\, T\, x \\
[\![\mathsf{ind}\, A\, F]\!]_{\mathsf{fun}}\, U\, T\, (g, x) := [\![F\,(T \circ g)]\!]_{\mathsf{fun}}\, U\, T\, x
\end{array}$$

Now, given a type D and a code $\gamma : \mathsf{IR}_D$, the inductive recursive definition of a type A_γ and a function T_γ is given as follows.

$$\begin{array}{ll} \mathsf{data}\,\mathsf{A}_\gamma : \mathsf{Set} & \mathsf{T}_\gamma : \mathsf{A}_\gamma \to D \\ \quad \mathsf{con} : [\![\gamma]\!]_{\mathsf{con}}\,\mathsf{A}_\gamma\,\mathsf{T}_\gamma \to \mathsf{A}_\gamma & \mathsf{T}_\gamma\,(\mathsf{con}_\gamma\,w) := [\![\gamma]\!]_{\mathsf{fun}}\,\mathsf{A}_\gamma\,\mathsf{T}_\gamma\,w \end{array}$$

The example with V and Elem are given by the following code.

$$\begin{aligned} c : \mathsf{IR}_{\mathsf{Set}} := \mathsf{nonind}\,\mathsf{Bool}\,\Big(&\lambda y.\mathsf{if}\,y\,\mathsf{then}\,\mathsf{nil}\,\bot\,\mathsf{else}\,\mathsf{ind}\,\top \\ &\qquad\qquad \big(\lambda a \to \mathsf{ind}\,(a\,\mathsf{tt})\,(\lambda b.\mathsf{nil}\,((x : a\,\mathsf{tt}) \to b\,x))\big)\Big) \end{aligned}$$

Applying $[\![-]\!]_{\mathsf{con}}$ on this code we get the following.

$$\begin{aligned} &[\![c]\!]_{\mathsf{con}}\,V\,Elem \\ =&\,\Sigma(y : \mathsf{Bool}) \\ &\quad .[\![\mathsf{if}\,y\,\mathsf{then}\,\mathsf{nil}\,\bot\,\mathsf{else}\,\mathsf{ind}\,\top\,(\lambda a.\mathsf{ind}\,(a\,\mathsf{tt})\,(\lambda b.\mathsf{nil}\,((x : a\,\mathsf{tt}) \to b\,x)))]\!]_{\mathsf{con}}\,V\,Elem \end{aligned}$$

If the y component is true then this computes to $\top$ which corresponds to the constructor zero which does not have any arguments. If it is false, it computes to

$$\Sigma(a : \top \to V).\Sigma\big(b : Elem\,(a\,\mathsf{tt}) \to V\big).\top$$

which corresponds to the $(a : \mathsf{V})$ and $(\mathsf{Elem}\,a \to \mathsf{V})$ arguments of the constructor pi.

Applying $[\![-]\!]_{\mathsf{fun}}$ on the code and the two different kinds of arguments of the constructor implements the function Elem.

$$\begin{aligned} [\![c]\!]_{\mathsf{fun}}\,V\,Elem\,(\mathsf{true},\mathsf{tt}) &= [\![\mathsf{nil}\,\bot]\!]_{\mathsf{fun}}\,V\,Elem\,\mathsf{tt} = \bot \\ [\![c]\!]_{\mathsf{fun}}\,V\,Elem\,(\mathsf{false},a,b,\mathsf{tt}) &= (x : Elem\,(a\,\mathsf{tt})) \to Elem\,(b\,x) \end{aligned}$$

A set theoretic model of the inductive recursive definitions given by the above codes is given in [50].

Small induction recursion is the special case of induction recursion when D is a small type. Such definitions can translated into inductive families [68]. In this thesis, we will only make use of small induction recursion.

2.3 Inductive inductive types

We denote mutually defined inductive types by first declaring the types and then separately listing the constructors for each. An example is the mutual definition of the predicates Odd and Even on natural numbers.

$$
\begin{array}{ll}
\mathsf{data}\,\mathsf{Even} & : \mathbb{N} \to \mathsf{Set} \\
\mathsf{data}\,\mathsf{Odd} & : \mathbb{N} \to \mathsf{Set} \\
\mathsf{data}\,\mathsf{Even} & \\
\quad \mathsf{zeroEven} & : \mathsf{Even}\,\mathsf{zero} \\
\quad \mathsf{sucOdd} & : (n : \mathbb{N}) \to \mathsf{Odd}\,n \to \mathsf{Even}\,(\mathsf{suc}\,n) \\
\mathsf{data}\,\mathsf{Odd} & \\
\quad \mathsf{sucEven} & : (n : \mathbb{N}) \to \mathsf{Even}\,n \to \mathsf{Odd}\,(\mathsf{suc}\,n)
\end{array}
$$

Even numbers are defined by the base case zero and saying that the successor of an odd number is even, while an odd number must be the successor of an even number.

Such definitions can be reduced to a single inductive type with an additional index of type Bool which says which original type was meant.

$$
\begin{array}{ll}
\mathsf{data}\,\mathsf{Even?} & : \mathbb{N} \to \mathsf{Bool} \to \mathsf{Set} \\
\quad \mathsf{zeroEven} & : \mathsf{Even?}\,\mathsf{zero}\,\mathsf{true} \\
\quad \mathsf{sucOdd} & : (n : \mathbb{N}) \to \mathsf{Even?}\,n\,\mathsf{false} \to \mathsf{Even?}\,(\mathsf{suc}\,n)\,\mathsf{true} \\
\quad \mathsf{sucEven} & : (n : \mathbb{N}) \to \mathsf{Even?}\,n\,\mathsf{true} \ \to \mathsf{Even?}\,(\mathsf{suc}\,n)\,\mathsf{false}
\end{array}
$$

$$
\begin{array}{l}
\mathsf{Even}\,(n : \mathbb{N}) : \mathsf{Set} := \mathsf{Even?}\,n\,\mathsf{true} \\
\mathsf{Odd}\ \ (n : \mathbb{N}) : \mathsf{Set} := \mathsf{Even?}\,n\,\mathsf{false}
\end{array}
$$

However there are certain mutual inductive definitions for which this method does not work. An example is the simultaneous definition of a type and a family of types indexed over the first type. These are called *inductive inductive types* (IITs) and we will need them to define the syntax of type theory with dependent types. Indeed, the typed syntax of type theory was the motivation for introducing the notion of inductive inductive types [79] and the main example is the following fragment from the definition of the syntax.

$$\begin{aligned}
&\mathsf{data}\,\mathsf{Con} &&: \mathsf{Set}\\
&\mathsf{data}\,\mathsf{Ty} &&: \mathsf{Con} \to \mathsf{Set}\\
&\mathsf{data}\,\mathsf{Con} &&\\
&\quad \cdot &&: \mathsf{Con}\\
&\quad -,- &&: (\Gamma : \mathsf{Con}) \to \mathsf{Ty}\,\Gamma \to \mathsf{Con}\\
&\mathsf{data}\,\mathsf{Ty} &&\\
&\quad \mathsf{U} &&: \mathsf{Ty}\,\Gamma\\
&\quad \Pi &&: (A : \mathsf{Ty}\,\Gamma) \to \mathsf{Ty}\,(\Gamma, A) \to \mathsf{Ty}\,\Gamma
\end{aligned}$$

Here Ty represents types and it is indexed by Con representing contexts. In contrast to the $\mathsf{Even\text{-}Odd}$ example, the order of declaring the types matter: Con needs to be in scope when declaring Ty. Then the constructors for each type are listed and later constructors can refer to earlier constructors as in the case of Π which refers to $-,-$ in its second argument. This example of IIT is made of two types but there is no limit in the number of constituent types.

Another example of an IIT is the type of sorted lists. This could be expressed without IITs, but the following definition seems to be a natural one. We define the type $\mathsf{SortedList}$ and the relation $- \leq_{\mathsf{SortedList}} -$ on natural numbers and sorted lists mutually.

$$\begin{aligned}
&\mathsf{data}\,\mathsf{SortedList} : \mathsf{Set}\\
&\mathsf{data}\,- \leq_{\mathsf{SortedList}} - : \mathbb{N} \to \mathsf{SortedList} \to \mathsf{Set}\\
&\mathsf{data}\,\mathsf{SortedList}\\
&\quad \mathsf{nil} \quad : \mathsf{SortedList}\\
&\quad \mathsf{cons} \quad : (x : \mathbb{N})(xs : \mathsf{SortedList}) \to x \leq_{\mathsf{SortedList}} xs \to \mathsf{SortedList}\\
&\mathsf{data}\,- \leq_{\mathsf{SortedList}} -\\
&\quad \mathsf{nil}_{\leq} \quad : x \leq_{\mathsf{SortedList}} \mathsf{nil}\\
&\quad \mathsf{cons}_{\leq} : y \leq x \to (p : x \leq_{\mathsf{SortedList}} xs) \to y \leq_{\mathsf{SortedList}} \mathsf{cons}\,x\,xs\,p
\end{aligned}$$

The cons constructor needs a natural number which is less than or equal to all the elements in the sorted list. $\mathsf{nil}_{\leq}$ expresses that any number is less than or equal to all the elements in an empty list. The $\mathsf{cons}_{\leq}$ constructor expresses that a y which is less than or equal to the first element x of a sorted list is less than or equal to all elements of that list.

The eliminator for an IIT needs as many motives as the number of constituent types and a method for each constructor. For our first example we will have two eliminators which depend on each other mutually, one for Con and one for Ty. As they share the arguments, we collect these into a record.

The record $\mathsf{MRecConTy}$ contains the motives and methods for the recursor.

$$\begin{array}{l}
\mathsf{record}\ \mathsf{MRecConTy} : \mathsf{Set}_1\\
\quad \begin{array}{ll}
\mathsf{Con}^{\mathsf{M}} & : \mathsf{Set}\\
\mathsf{Ty}^{\mathsf{M}} & : \mathsf{Con}^{\mathsf{M}} \to \mathsf{Set}\\
\cdot^{\mathsf{M}} & : \mathsf{Con}^{\mathsf{M}}\\
-,^{\mathsf{M}}- & : (\Gamma^{M} : \mathsf{Con}^{\mathsf{M}}) \to \mathsf{Ty}^{\mathsf{M}}\,\Gamma^{M} \to \mathsf{Con}^{\mathsf{M}}\\
\mathsf{U}^{\mathsf{M}} & : \mathsf{Ty}^{\mathsf{M}}\,\Gamma^{\mathsf{M}}\\
\Pi^{\mathsf{M}} & : (A^{M} : \mathsf{Ty}^{\mathsf{M}}\,\Gamma^{M}) \to \mathsf{Ty}^{\mathsf{M}}\,(\Gamma^{M},^{\mathsf{M}}A^{M}) \to \mathsf{Ty}^{\mathsf{M}}\,\Gamma^{M}
\end{array}
\end{array}$$

The types of the methods reflect the types of the constructors but they refer to the motives and the previous methods by putting the ^{M} indices everywhere.

The recursor is given in a module parameterised by the above record and all fields of the record are made visible by opening it. We declare the recursors RecCon and RecTy and then list their computation rules. This can also be viewed as a *definition* of the recursor by pattern matching (and indeed this is how we reproduce the recursor in Agda).

$$\begin{array}{l}
\mathsf{module}\ \mathsf{RecConTy}\,(m : \mathsf{MRecConTy})\\
\quad \mathsf{open}\ \mathsf{MRecConTy}\,m\\
\quad \begin{array}{lll}
\mathsf{RecCon} & : \mathsf{Con} & \to \mathsf{Con}^{\mathsf{M}}\\
\mathsf{RecTy} & : \mathsf{Ty}\,\Gamma & \to \mathsf{Ty}^{\mathsf{M}}\,(\mathsf{RecCon}\,\Gamma)\\
\mathsf{RecCon} & \cdot & = \cdot^{\mathsf{M}}\\
\mathsf{RecCon} & (\Gamma, A) & = (\mathsf{RecCon}\,\Gamma),^{\mathsf{M}}(\mathsf{RecTy}\,A)\\
\mathsf{RecTy} & \mathsf{U} & = \mathsf{U}^{\mathsf{M}}\\
\mathsf{RecTy} & (\Pi\,A\,B) & = \Pi^{\mathsf{M}}\,(\mathsf{RecTy}\,A)\,(\mathsf{RecTy}\,B)
\end{array}
\end{array}$$

The record $\mathsf{MElimConTy}$ contains the motives and methods for the eliminator.

The motives are families over Con and Ty and the methods are elements of these families at the corresponding constructor. An algorithm for computing the types of the motives and methods is given in section 4.3.

$$
\begin{aligned}
&\mathsf{record}\,\mathsf{MElimConTy} : \mathsf{Set}_1\\
&\quad \mathsf{Con}^{\mathsf{M}} && : \mathsf{Con} \to \mathsf{Set}\\
&\quad \mathsf{Ty}^{\mathsf{M}} && : \mathsf{Con}^{\mathsf{M}}\,\Gamma \to \mathsf{Ty}\,\Gamma \to \mathsf{Set}\\
&\quad \cdot^{\mathsf{M}} && : \mathsf{Con}^{\mathsf{M}}\,\cdot\\
&\quad -,^{\mathsf{M}}- && : (\Gamma^{M} : \mathsf{Con}^{\mathsf{M}}\,\Gamma) \to \mathsf{Ty}^{\mathsf{M}}\,\Gamma^{M}\,A \to \mathsf{Con}^{\mathsf{M}}\,(\Gamma, A)\\
&\quad \mathsf{U}^{\mathsf{M}} && : \mathsf{Ty}^{\mathsf{M}}\,\Gamma^{M}\,\mathsf{U}\\
&\quad \Pi^{\mathsf{M}} && : (A^{M} : \mathsf{Ty}^{\mathsf{M}}\,\Gamma^{M}\,A) \to \mathsf{Ty}^{\mathsf{M}}\,(\Gamma^{M},^{\mathsf{M}}A^{M})\,B \to \mathsf{Ty}^{\mathsf{M}}\,\Gamma^{M}\,(\Pi\,A\,B)
\end{aligned}
$$

If we write down the implicit quantifications, the type of Π^{M} is the following.

$$
\begin{aligned}
&\{\Gamma : \mathsf{Con}\}\{\Gamma^{M} : \mathsf{Con}^{\mathsf{M}}\,\Gamma\}\{A : \mathsf{Ty}\,\Gamma\}(A^{M} : \mathsf{Ty}^{\mathsf{M}}\,\Gamma^{M}\,A)\{B : \mathsf{Ty}\,(\Gamma, A)\}\\
&\to \mathsf{Ty}^{\mathsf{M}}\,(\Gamma^{M},^{\mathsf{M}}A^{M})\,B \to \mathsf{Ty}^{\mathsf{M}}\,\Gamma^{M}\,(\Pi\,A\,B)
\end{aligned}
$$

The eliminators have dependent types and the computation rules are the same as for the recursor.

$$
\begin{aligned}
&\mathsf{module}\,\mathsf{ElimConTy}\,(m : \mathsf{MElimConTy})\\
&\quad \mathsf{open}\,\mathsf{MElimConTy}\,m\\
&\quad \mathsf{ElimCon} && : (\Gamma : \mathsf{Con}) && \to \mathsf{Con}^{\mathsf{M}}\,\Gamma\\
&\quad \mathsf{ElimTy} && : (A : \mathsf{Ty}\,\Gamma) && \to \mathsf{Ty}^{\mathsf{M}}\,(\mathsf{ElimCon}\,\Gamma)\,A\\
&\quad \mathsf{ElimCon}\ \cdot && && = \cdot^{\mathsf{M}}\\
&\quad \mathsf{ElimCon}\ (\Gamma, A) && && = (\mathsf{ElimCon}\,\Gamma),^{\mathsf{M}}(\mathsf{ElimTy}\,A)\\
&\quad \mathsf{ElimTy}\ \ \mathsf{U} && && = \mathsf{U}^{\mathsf{M}}\\
&\quad \mathsf{ElimTy}\ \ (\Pi\,A\,B) && && = \Pi^{\mathsf{M}}\,(\mathsf{ElimTy}\,A)\,(\mathsf{ElimTy}\,B)
\end{aligned}
$$

We describe the recursor of $\mathsf{SortedList}$. First we collect the arguments into the record $\mathsf{MRecSortedList}$.

$$
\begin{array}{ll}
\multicolumn{2}{l}{\mathsf{record\ MRecSortedList} : \mathsf{Set}_1} \\
\quad \mathsf{SortedList}^{\mathsf{M}} & : \mathsf{Set} \\
\quad - \leq^{\mathsf{M}}_{\mathsf{SortedList}} - & : \mathbb{N} \to \mathsf{SortedList}^{\mathsf{M}} \to \mathsf{Set} \\
\quad \mathsf{nil}^{\mathsf{M}} & : \mathsf{SortedList}^{\mathsf{M}} \\
\quad \mathsf{cons}^{\mathsf{M}} & : (x : \mathbb{N})(xs^{M} : \mathsf{SortedList}^{\mathsf{M}}) \to x \leq^{\mathsf{M}}_{\mathsf{SortedList}} xs^{M} \to \mathsf{SortedList}^{\mathsf{M}} \\
\quad \mathsf{nil}^{\mathsf{M}}_{\leq} & : x \leq^{\mathsf{M}}_{\mathsf{SortedList}} \mathsf{nil}^{\mathsf{M}} \\
\quad \mathsf{cons}^{\mathsf{M}}_{\leq} & : y \leq x \to (p^{M} : x \leq^{\mathsf{M}}_{\mathsf{SortedList}} xs^{M}) \to y \leq^{\mathsf{M}}_{\mathsf{SortedList}} \mathsf{cons}^{\mathsf{M}}\, x\, xs^{M}\, p^{\mathsf{M}}
\end{array}
$$

The recursor is given by the following functions and computation rules.

$$
\begin{array}{lll}
\multicolumn{3}{l}{\mathsf{record\ RecSortedList}(m : \mathsf{MRecSortedList}) : \mathsf{Set}_1} \\
\multicolumn{3}{l}{\quad \mathsf{open\ MRecSortedList}\, m} \\
\quad \mathsf{RecSortedList} & : \mathsf{SortedList} & \to \mathsf{SortedList}^{\mathsf{M}} \\
\quad \mathsf{Rec}{\leq_{\mathsf{SortedList}}} & : x \leq_{\mathsf{SortedList}} xs & \to x \leq^{\mathsf{M}}_{\mathsf{SortedList}} \mathsf{RecSortedList}\, xs \\
\quad \mathsf{RecSortedList}\, \mathsf{nil} & & = \mathsf{nil}^{\mathsf{M}} \\
\quad \mathsf{RecSortedList}\, (\mathsf{cons}\, x\, xs\, p) & & = \mathsf{cons}^{\mathsf{M}}\, x\, (\mathsf{RecSortedList})\, (\mathsf{Rec}{\leq_{\mathsf{SortedList}}}\, p) \\
\quad \mathsf{Rec}{\leq_{\mathsf{SortedList}}}\ \mathsf{nil}_{\leq} & & = \mathsf{nil}^{\mathsf{M}}_{\leq} \\
\quad \mathsf{Rec}{\leq_{\mathsf{SortedList}}}\ (\mathsf{cons}_{\leq}\, w\, p) & & = \mathsf{cons}^{\mathsf{M}}_{\leq}\, w\, (\mathsf{Rec}{\leq_{\mathsf{SortedList}}}\, p)
\end{array}
$$

2.3.1 Codes for inductive inductive types

The constructors of inductive inductive types need to obey strict positivity conditions. Just as in the case of simple inductive types, inductive families and induction recursion, the inductive inductive types which are allowed can be given using codes together with decoding functions [79]. The specification of codes for an inductive inductive type which is built up by a type A and a family $\mathsf{B} : \mathsf{A} \to \mathsf{Set}$ is given as follows. The codes for A and B are given by C_{A} and C_{B}. We denote the decoding functions by $[\![-]\!]_{\mathsf{A}}$ and $[\![-]\!]_{\mathsf{B}}$. C_{B} depends on the code for A. The decoding functions need a set A and a family B over A. Furthermore, the decoding function for B needs the constructor of A. The reason for this is that the constructor of B can reference the constructor of A, as in the case of Π and $\mathsf{cons}_{\leq}$

above. The function index calculates the index of B given an argument of the constructor.

$$
\begin{aligned}
&\mathsf{C_A} && : \mathsf{Set}\\
&[\![-]\!]_\mathsf{A} && : \mathsf{C_A} \to (A : \mathsf{Set}) \to (A \to \mathsf{Set}) \to \mathsf{Set}\\
&\mathsf{C_B} && : \mathsf{C_A} \to \mathsf{Set}\\
&[\![-]\!]_\mathsf{B} && : \{\gamma : \mathsf{C_A}\} \to \mathsf{C_B}\,\gamma \to (A : \mathsf{Set})(B : A \to \mathsf{Set})(con_A : [\![\gamma]\!]_\mathsf{A}\, A\, B \to A) \to \mathsf{Set}\\
&\mathsf{index} : [\![\delta]\!]_\mathsf{B}\, A\, B\, con_A \to A
\end{aligned}
$$

The codes $\gamma : \mathsf{C_A}$ and $\delta : \mathsf{C_B}\,\gamma$ encode the following inductive type.

$$
\begin{aligned}
&\mathsf{data}\ \mathsf{A}_{\gamma\,\delta} : \mathsf{Set}\\
&\mathsf{data}\ \mathsf{B}_{\gamma\,\delta} : \mathsf{A}_{\gamma\,\delta} \to \mathsf{Set}\\
&\quad \mathsf{con_A} : [\![\gamma_A]\!]_\mathsf{A}\, \mathsf{A}_{\gamma\,\delta}\, \mathsf{B}_{\gamma\,\delta} \to \mathsf{A}_{\gamma\,\delta}\\
&\quad \mathsf{con_B} : (w : [\![\delta]\!]_\mathsf{B}\, \mathsf{A}_{\gamma\,\delta}\, \mathsf{B}_{\gamma\,\delta}\, \mathsf{con_A}) \to \mathsf{B}_{\gamma\,\delta}\, (\mathsf{index}\, w)
\end{aligned}
$$

Now we will describe how the above codes and decoding functions are defined. We follow the construction in section 3.2.3 of [79].

As in the case of codes for induction recursion, the codes $\mathsf{C_A}$ are given as an inductive type and we have separate constructors for non-inductive and inductive arguments and for the base case. Also, we add another argument A_{ref} to the type which collects together the inductive arguments that one can refer to in later arguments of the constructor. Hence, the codes are given by the inductive type $\mathsf{C'_A}$ and $\mathsf{C_A} := \mathsf{C'_A}\, \bot$.

$$
\begin{aligned}
&\mathsf{data}\ \mathsf{C'_A}\, (A_{ref} : \mathsf{Set}) : \mathsf{Set}_1\\
&\quad \mathsf{nil} && : \mathsf{C'_A}\, A_{ref}\\
&\quad \mathsf{nonind} && : (K : \mathsf{Set}) \to (K \to \mathsf{C'_A}\, A_{ref}) \to \mathsf{C'_A}\, A_{ref}\\
&\quad \mathsf{ind_A} && : (K : \mathsf{Set}) \to \mathsf{C'_A}\, (A_{ref} + K) \to \mathsf{C'_A}\, A_{ref}\\
&\quad \mathsf{ind_B} && : (K : \mathsf{Set})(h_{index} : K \to A_{ref}) \to \mathsf{C'_A}\, A_{ref} \to \mathsf{C'_A}\, A_{ref}
\end{aligned}
$$

We have the code nil for the trivial constructor, nonind for a non-inductive argument on which the rest of the constructors can depend, $\mathsf{ind_A}$ for an inductive argument of type A which extends the set A_{ref}, and finally $\mathsf{ind_B}$ for an inductive argument of type B which does not extend A_{ref} but we need the A_{ref}-index of this argument. This is given by h_{index}.

For example, the code for the constructor $-,-$ in the Con-Ty example can be given as follows.

$$c_{-,-} : \mathsf{C}'_{\mathsf{A}}\,\bot := \mathsf{ind}_{\mathsf{A}}\,\top\,(\mathsf{ind}_{\mathsf{B}}\,\top\,(\lambda\_ \to \mathsf{inj}_2\,\mathsf{tt})\,\mathsf{nil})$$

The code for $\cdot$ is the simplest possible one.

$$c_{\cdot} : \mathsf{C}'_{\mathsf{A}}\,\bot := \mathsf{nil}$$

The decoding function for A is defined using the following more general decoding function: $[\![-]\!]_{\mathsf{A}}\,\gamma\,A\,B := [\![-]\!]'_{\mathsf{A}}\,\{\bot\}\,\gamma\,A\,B\,\mathsf{Elim}\bot$. The extra argument rep_A provides the representation of the elements of A_{ref} in A.

$$\begin{aligned}
&[\![-]\!]'_{\mathsf{A}} : \{A_{ref} : \mathsf{Set}\}(\gamma : \mathsf{C}'_{\mathsf{A}}\,A_{ref})(A : \mathsf{Set})(B : A \to \mathsf{Set})(rep_A : A_{ref} \to A) \to \mathsf{Set}\\
&[\![\mathsf{nil}]\!]'_{\mathsf{A}}\,\_\,\_\,\_ := \top\\
&[\![\mathsf{nonind}\,K\,\gamma]\!]'_{\mathsf{A}}\,A\,B\,rep_A := \Sigma(k : K).[\![\gamma\,k]\!]'_{\mathsf{A}}\,A\,B\,rep_A\\
&[\![\mathsf{ind}_{\mathsf{A}}\,K\,\gamma]\!]'_{\mathsf{A}}\,A\,B\,rep_A := \Sigma(j : K \to A).[\![\gamma]\!]'_{\mathsf{A}}\,A\,B\,[rep_A, j]\\
&[\![\mathsf{ind}_{\mathsf{B}}\,K\,h_{index}\,\gamma]\!]'_{\mathsf{A}}\,A\,B\,rep_A := \big((k : K) \to B\,(rep_A\,(h_{index}\,k))\big) \times \big([\![\gamma]\!]'_{\mathsf{A}}\,A\,B\,rep_A\big)
\end{aligned}$$

Now we can compute

$$\begin{aligned}
&[\![c_{-,-}]\!]_{\mathsf{A}}\,Con\,Ty\\
=&\,\Sigma(\Gamma : \top \to Con).[\![\mathsf{ind}_{\mathsf{B}}\,\top\,(\lambda\,\mathsf{tt}.\mathsf{inj}_2\,\mathsf{tt})\,\mathsf{nil}]\!]'_{\mathsf{A}}\,Con\,Ty\,[\mathsf{Elim}\bot, \Gamma]\\
=&\,\Sigma(\Gamma : \top \to Con).((\_ : \top) \to Ty\,(\Gamma\,\mathsf{tt})) \times \top,
\end{aligned}$$

which represents the arguments $(\Gamma : \mathsf{Con})$ and $\mathsf{Ty}\,\Gamma$ of $-,-$. Decoding $c_{\cdot}$ gives $\top$.

The argument of the constructor of B can refer to the constructors of A (see Π in the Con-Ty example), hence we need to represent the usages of the constructor of A in the codes of type B. The constructor of A can refer to terms of type B, so we need to refer to these as well. This is why we define a type of codes T_{A} for terms of type A together with a decoding function $[\![-]\!]_{\mathsf{T}_{\mathsf{A}}}$ and similarly for B.

T_{A} and T_{B} are given as an inductive recursive definition. Similarly to A_{ref}, we use a set B_{ref} for the referrable inductive arguments of B. The three constructors of T_{A} encode the referrable terms in A_{ref}, the A-index of a term in B_{ref}, and a usage of the constructor of A, respectively. We can only encode terms of type B

when they are referred to in B_{ref}, hence the definition of $\mathsf{T_B}$.

$$\begin{array}{ll}
\mathsf{data}\,\mathsf{T_A}(A_{ref}\,B_{ref} : \mathsf{Set})(\gamma : \mathsf{C_A}) : \mathsf{Set} & \mathsf{T_B} : \mathsf{T_A}\,A_{ref}\,B_{ref}\,\gamma \to \mathsf{Set} \\
\quad \mathsf{aref} : A_{ref} \to \mathsf{T_A}\,A_{ref}\,B_{ref}\,\gamma & \mathsf{T_B}\,(\mathsf{aref}\,a) := \bot \\
\quad \mathsf{bref} : B_{ref} \to \mathsf{T_A}\,A_{ref}\,B_{ref}\,\gamma & \mathsf{T_B}\,(\mathsf{bref}\,b) := \top \\
\quad \mathsf{arg} : \llbracket\gamma\rrbracket_{\mathsf{A}}\,(\mathsf{T_A}\,A_{ref}\,B_{ref}\,\gamma)\,\mathsf{T_B} \to \mathsf{T_A}\,A_{ref}\,B_{ref}\,\gamma & \mathsf{T_B}\,(\mathsf{arg}\,x) := \bot
\end{array}$$

We define the decoding functions mutually. They take as arguments not only the types A, the family $B : A \to \mathsf{Set}$ and the constructor of A, but also functions representing the referenced arguments. rep_A is like before, rep_{index} gives the A-index and rep_B creates the object in the family at the index given by rep_{index}.

$$\begin{array}{ll}
\llbracket - \rrbracket_{\mathsf{T_A}} & : (a : \mathsf{T_A}\,A_{ref}\,B_{ref}\gamma)(A : \mathsf{Set})(B : A \to \mathsf{Set})(rep_A : A_{ref} \to A) \\
& \quad (rep_{index} : B_{ref} \to A)(rep_B : (b : B_{ref}) \to B\,(rep_{index}\,b)) \\
& \quad (con_A : \llbracket\gamma\rrbracket_{\mathsf{A}}\,A\,B \to A) \to A \\
\llbracket -, - \rrbracket_{\mathsf{T_B}} & : (a : \mathsf{T_A}\,A_{ref}\,B_{ref}\gamma)(b : \mathsf{T_B}\,a)(A : \mathsf{Set})(B : A \to \mathsf{Set})(rep_A : A_{ref} \to A) \\
& \quad (rep_{index} : B_{ref} \to A)(rep_B : (b : B_{ref}) \to B\,(rep_{index}\,b)) \\
& \quad (con_A : \llbracket\gamma\rrbracket_{\mathsf{A}}\,A\,B \to A) \to B\,\big(\llbracket a\rrbracket_{\mathsf{T_A}}\,A\,B\,rep_A\,rep_{index}\,rep_B\,con_A\big)
\end{array}$$

$$\begin{array}{l}
\llbracket \mathsf{aref}\,a \rrbracket_{\mathsf{T_A}}\,\_\,\_\,rep_A\,\_\,\_\,\_ := rep_A\,a \\
\llbracket \mathsf{bref}\,b \rrbracket_{\mathsf{T_A}}\,\_\,\_\,\_\,rep_{ind}\,\_\,\_ := rep_{ind}\,b \\
\llbracket \mathsf{arg}\,x \rrbracket_{\mathsf{T_A}}\,\_\,\_\,\_\,rep_{ind}\,\_\,con_A := con_A\,\big(\llbracket\gamma\rrbracket_{\mathsf{A}}\mathsf{func}\,(\llbracket-\rrbracket_{\mathsf{T_A}}\,...)\,(\llbracket-\rrbracket_{\mathsf{T_B}}\,...)\,x\big) \\
\llbracket \mathsf{aref}\,a, w \rrbracket_{\mathsf{T_B}}\,\_\,\_\,\_\,\_\,\_\,\_ := \mathsf{Elim}\bot\,w \\
\llbracket \mathsf{bref}\,b, \mathsf{tt} \rrbracket_{\mathsf{T_B}}\,\_\,\_\,\_\,\_\,rep_B\,\_ := rep_B\,b \\
\llbracket \mathsf{arg}\,a, w \rrbracket_{\mathsf{T_B}}\,\_\,\_\,\_\,\_\,\_\,\_ := \mathsf{Elim}\bot\,w
\end{array}$$

The arg case of $\llbracket-\rrbracket_{\mathsf{T_A}}$ uses functoriality of $\llbracket-\rrbracket_{\mathsf{A}}$, see the formalisation for details.

With the help of these we give the definition of the codes for B, the decoding function and the indexing function. As in the case of $\mathsf{C_A}$, we define generalised versions indexed by a set of references. The codes for B give either a constructor with no arguments (in this case we have to specify the index of the constructor which is a term of type A), a non inductive argument, an inductive argument of type A (in this case we extend A_{ref}) or an inductive argument of type B (in this case h_{index} says what the index of the argument is and we extend B_{ref}).

$$
\begin{array}{lll}
\multicolumn{3}{l}{\mathsf{data}\ \mathsf{C}'_{\mathsf{B}}(A_{ref}\,B_{ref} : \mathsf{Set})(\gamma : \mathsf{C}_{\mathsf{A}}) : \mathsf{Set}_1} \\
\quad \mathsf{nil} & : \mathsf{T}_{\mathsf{A}}\,A_{ref}\,B_{ref}\,\gamma \to \mathsf{C}'_{\mathsf{B}}\,A_{ref}\,B_{ref}\,\gamma \\
\quad \mathsf{nonind} & : (K : \mathsf{Set}) \to (K \to \mathsf{C}'_{\mathsf{B}}\,A_{ref}\,B_{ref}\,\gamma) \to \mathsf{C}'_{\mathsf{B}}\,A_{ref}\,B_{ref}\,\gamma \\
\quad \mathsf{indA} & : (K : \mathsf{Set}) \to \mathsf{C}'_{\mathsf{B}}\,(A_{ref} + K)\,B_{ref}\,\gamma \to \mathsf{C}'_{\mathsf{B}}\,A_{ref}\,B_{ref}\,\gamma \\
\quad \mathsf{indB} & : (K : \mathsf{Set})(h_{index} : K \to \mathsf{T}_{\mathsf{A}}\,A_{ref}\,B_{ref}\,\gamma) \to \mathsf{C}'_{\mathsf{B}}\,A_{ref}\,(B_{ref} + K)\,\gamma \\
 & \to \mathsf{C}'_{\mathsf{B}}\,A_{ref}\,B_{ref}\,\gamma
\end{array}
$$

The decoding of C'_{B} follows the above explanation about the codes.

$$
\begin{array}{l}
\llbracket - \rrbracket'_{\mathsf{B}} : (\delta : \mathsf{C}'_{\mathsf{B}}\,A_{ref}\,B_{ref}\,\gamma)(A : \mathsf{Set})(B : A \to \mathsf{Set}) \\
\qquad (rep_A : A_{ref} \to A)(rep_{index} : B_{ref} \to A)(rep_B : (b : B_{ref}) \to B\,(rep_{index}\,b)) \\
\qquad (con_A : \llbracket \gamma \rrbracket_{\mathsf{A}}\,A\,B \to A) \to \mathsf{Set} \\
\llbracket \mathsf{nil}\,\_ \rrbracket'_{\mathsf{B}}\,\_\,\_\,\_\,\_\,\_\,\_ := \top \\
\llbracket \mathsf{nonind}\,K\,\delta \rrbracket'_{\mathsf{B}}\,A\,B\,rep_A\,rep_{index}\,rep_B\,con_A \\
\qquad := \Sigma(k : K).\llbracket \delta\,k \rrbracket'_{\mathsf{B}}\,A\,B\,rep_A\,rep_{index}\,rep_B\,con_A \\
\llbracket \mathsf{indA}\,K\,\delta \rrbracket'_{\mathsf{B}}\,A\,B\,rep_A\,rep_{index}\,rep_B\,con_A \\
\qquad := \Sigma\big(j : (k : K) \to A\big).\llbracket \delta \rrbracket'_{\mathsf{B}}\,A\,B\,[rep_A, j]\,rep_{index}\,rep_B\,con_A \\
\llbracket \mathsf{indB}\,K\,h_{index}\,\delta \rrbracket'_{\mathsf{B}}\,A\,B\,rep_A\,rep_{index}\,rep_B\,con_A \\
\qquad := \Sigma\big(j : (k : K) \to B\,(\llbracket h_{index}\,k \rrbracket_{\mathsf{T_A}}\,A\,B\,rep_A\,rep_{index}\,rep_B\,con_A)\big) \\
\qquad\quad .\llbracket \delta \rrbracket'_{\mathsf{B}}\,A\,B\,rep_A\,[rep_{index}, \lambda k.\llbracket h_{index}\,k \rrbracket_{\mathsf{T_A}} ...]\,[rep_B, j]\,con_A
\end{array}
$$

The index of a constructor is calculated from the code in the nil case using the interpretation of terms of type A. The other cases look up the result recursively.

$$
\begin{array}{lll}
\multicolumn{3}{l}{\mathsf{index}' : (\delta : \mathsf{C}'_{\mathsf{B}}\,A_{ref}\,B_{ref}\,\gamma) \to \llbracket \delta \rrbracket'_{\mathsf{B}}\,A\,B\,rep_A\,rep_{index}\,rep_B\,con_A \to A} \\
\mathsf{index}'\,(\mathsf{nil}\,a) & \mathsf{tt} & := \llbracket a \rrbracket_{\mathsf{T_A}}\,A\,B\,rep_A\,rep_{index}\,rep_B\,con_A \\
\mathsf{index}'\,(\mathsf{nonind}\,K\,\delta) & (k, y) & := \mathsf{index}'\,(\delta\,k)\,y \\
\mathsf{index}'\,(\mathsf{indA}\,K\,\delta) & (j, y) & := \mathsf{index}'\,\delta\,y \\
\mathsf{index}'\,(\mathsf{indB}\,K\,h_{index}\,\delta) & (j, y) & := \mathsf{index}'\,\delta\,y
\end{array}
$$

The final versions of the codes for B, the decoding and indexing functions are using the special cases of the above when A_{ref} and B_{ref} are both the empty type.

$$\begin{aligned}
&\mathsf{C_B}\,(\gamma : \mathsf{C_A}) : \mathsf{Set} := \mathsf{C'_B}\,\bot\,\bot\,\gamma \\
&[\![\delta]\!]_{\mathsf{B}}\,A\,B\,con_A : \mathsf{Set} := [\![\delta]\!]'_{\mathsf{B}}\,A\,B\ \mathsf{Elim}\bot\ \mathsf{Elim}\bot\ \mathsf{Elim}\bot\ con_A \\
&\mathsf{index}\,(b : [\![\delta]\!]_{\mathsf{B}}\,A\,B\,con_A) : A := \mathsf{index}'\,\delta\,b
\end{aligned}$$

Now we show how to encode the constructors of Ty in the $\mathsf{Con\text{-}Ty}$ example.

$$\begin{aligned}
c_{\mathsf{U}} &: \mathsf{C_B}\,c_{-,-} := \mathsf{ind_A}\,\top\,\big(\mathsf{nil}\,(\mathsf{aref}\,(\mathsf{inj_2}\,\mathsf{tt}))\big) \\
c_{\Pi} &: \mathsf{C_B}\,c_{-,-} := \mathsf{ind_A}\,\top\,\Big(\mathsf{ind_B}\,\top\,\big(\lambda_{-}.\mathsf{aref}\,(\mathsf{inj_2}\,\mathsf{tt})\big) \\
&\qquad\qquad\big(\mathsf{ind_B}\,\top\,(\lambda\,_{-}.\mathsf{arg}\,((\lambda_{-}.\mathsf{bref}\,(\mathsf{inj_2}\,\mathsf{tt})),((\lambda_{-}.\mathsf{tt}),\mathsf{tt}))) \\
&\qquad\qquad\qquad(\mathsf{nil}\,(\mathsf{aref}\,(\mathsf{inj_2}\,\mathsf{tt})))\big)\Big)
\end{aligned}$$

The constructor U only has one inductive argument of A ($\mathsf{ind_A}$) and returns that as an index (aref). The constructor Π has an inductive argument Γ of A which we will later refer to as $\mathsf{aref}\,(\mathsf{inj_2}\,\mathsf{tt})$. Then it takes an inductive argument A of type B at index Γ as expressed by $\mathsf{ind_B}\,\top\,(\lambda_{-}.\mathsf{aref}\,(\mathsf{inj_2}\,\mathsf{tt}))$. The final inductive argument is again of type B, but at the index Γ, A where the usage of $-,-$ is encoded by arg, the argument Γ is given by bref (this is the A-index of the argument of type B) and the argument of type B is given by $\lambda_{-}.\mathsf{tt}$ (there is only one element in B_{ref} at this point).

When decoding c_{U}, we get that $[\![c_{\mathsf{U}}]\!]_{\mathsf{B}}\,Con\,Ty\,con_{-,-} = \Sigma(\Gamma : \top \to Con).\top$ and

$$\begin{aligned}
[\![c_{\Pi}]\!]_{\mathsf{B}}\,Con\,Ty\,con_{-,-} = \;&\Sigma(\Gamma : \top \to Con) \\
&.\Sigma(A : \top \to Ty\,(\Gamma\,\mathsf{tt})) \\
&.\Sigma(B : \top \to Ty\,(con_{-,-}\,(\lambda_{-}.\Gamma\,\mathsf{tt},(\lambda_{-}.A\,\mathsf{tt},\mathsf{tt})))).\top
\end{aligned}$$

When applying the index function to elements of these types, one gets the expected results:

$$\begin{aligned}
&\mathsf{index}\,\{c_{\mathsf{U}}\}\,(\Gamma,\mathsf{tt}) &&= \Gamma\,\mathsf{tt} \\
&\mathsf{index}\,\{c_{\Pi}\}\,(\Gamma,A,B,\mathsf{tt}) &&= \Gamma\,\mathsf{tt}
\end{aligned}$$

The above presented codes for inductive inductive types can express the Con-

Ty example and need some generalisation to express the sorted list example (it doesn't allow the index of type $\mathbb{N}$ in the second type). In chapter 3 we will use more constituent types than two, i.e. a tower of inductive definitions like $A : \mathsf{Set}, B : A \to \mathsf{Set}, C : (a : A) \to B\,A \to \mathsf{Set}, ...$ The extensions of the above code system to these cases are described in section 6.2 of [79]. A categorical characterisation and a set-theoretic semantics of inductive inductive types is given in [20,79]. From a computational point of view IITs are not problematic and they are supported by Agda.

2.4 Quotient inductive types

Higher inductive types come from homotopy type theory (chapter 6 of [88]). They are a generalisation of inductive types: in addition to usual (point) constructors they allow the definition of equality constructors. A simple example is the higher inductive type of the interval.

$$
\begin{aligned}
&\mathsf{data}\,\mathsf{I} && : \mathsf{Set} \\
&\quad \mathsf{left} && : \mathsf{I} \\
&\quad \mathsf{right} && : \mathsf{I} \\
&\quad \mathsf{segment} && : \mathsf{left} \equiv \mathsf{right}
\end{aligned}
$$

This type has two usual constructors left and right and it has an equality constructor $\mathsf{segment}$ which adds an element to the identity type of I stating that left and right are equal. To eliminate from this type one needs three methods, two corresponding to the constructors left and right, and one corresponding to $\mathsf{segment}$. The method corresponding to $\mathsf{segment}$ ensures that the objects to which left and right are mapped to are equal. We list the recursor and the eliminator below. Note that in the case of the dependent eliminator this equality lives over the constructor $\mathsf{segment}$.

$$
\begin{aligned}
\mathsf{RecI} : \,&(I^M : \mathsf{Set}) \\
&(left^M\, right^M : I^M) \\
&(segment^M : left^M \equiv right^M) \\
&\to \mathsf{I} \to I^M
\end{aligned}
\qquad
\begin{aligned}
\mathsf{ElimI} : \,&(I^M : \mathsf{I} \to \mathsf{Set}) \\
&(left^M : I^M\,\mathsf{left})(right^M : I^M\,\mathsf{right}) \\
&(segment^M : left^M \equiv^{\mathsf{segment}} right^M) \\
&(i : \mathsf{I}) \to I^M\, i
\end{aligned}
$$

The computation rules for the point constructors are the usual ones, we list them for the recursor.

$$\begin{aligned}
\mathsf{RecI}\, I^M\, left^M\, right^M\, segment^M\, \mathsf{left} &= left^M \\
\mathsf{RecI}\, I^M\, left^M\, right^M\, segment^M\, \mathsf{right} &= right^M
\end{aligned}$$

The computation rule for $\mathsf{segment}$ expressed as a propositional equality states the following.

$$\mathsf{ap}\,(\mathsf{RecI}\, I^M\, left^M\, right^M\, segment^M)\,\mathsf{segment} \equiv segment^M$$

However, as we work in a strict metatheory (K is true), this equality is always true, hence there is no need to state it separately.

We call quotient inductive types (QITs) the higher inductive types in a strict theory (like ours). A consequence of K is that equalities between equalities are always trivial, this is why we don't use the term "higher". Alternatively, if we worked in homotopy type theory, quotient inductive types would mean set-truncated higher inductive types.

Functional extensionality is the fact that pointwise equal functions are equal:

$$\mathsf{funext} : \{f\, g : (x : A) \to B\, x\} \to \big((x : A) \to f\, x \equiv g\, x\big) \to f \equiv g.$$

We use this axiom throughout this thesis. It also follows from the existence of the interval quotient inductive type I. For details, see the formalisation.

QITs are not the same as quotient types [57, 65]. The latter can be seen as special cases of the former. For quotient types, one needs to define the type and the equivalence relation in separate steps; QITs give the opportunity to do these two things at the same time. Sometimes this

. In [88], there are two examples of such usage of : the constructable hierarchy of sets in an encoding of set theory and the definition of the real numbers as Cauchy-sequences. These definitions would have been possible using quotient types however it seems that they would have required some form of axiom of choice to be useful.

We give another example which points out the difference between quotient types and quotient inductive types.

Given a type and a binary relation over it, we can define the quotiented type by the following rules.[2]

$$
\begin{array}{ll}
\mathsf{data}\,{-}/{-} & (A : \mathsf{Set})(R : A \to A \to \mathsf{Set}) : \mathsf{Set} \\
\quad [-] & : A \to A/R \\
\quad [-]{\equiv} & : R\,a\,a' \to [a] \equiv [a'] \\
\mathsf{Rec}{-}/{-} & : (A : \mathsf{Set})(R : A \to A \to \mathsf{Set}) \\
 & (Q^M : \mathsf{Set}) \\
 & ([-]^M : A \to Q^M) \\
 & ([-]{\equiv}^M : R\,a\,a' \to [a]^M \equiv [a']^M) \\
 & \to A/R \to Q^M \\
\mathsf{Rec}A/R & Q^M\,[-]^M\,[-]{\equiv}^M\,[a] = [a]^M
\end{array}
$$

The recursor expresses that we can eliminate from the quotient into a type Q^M by providing a function $[-]^M$ from A to Q^M. However there is a limitation: this function needs to respect the relation R (this fact is witnessed by the method $[-]{\equiv}^M$).

For our example, first we will define binary trees quotiented by a relation which expresses that the order of subtrees does not matter.

$$
\begin{array}{ll}
\mathsf{data}\,\mathsf{T}_2 & : \mathsf{Set} \\
\quad \mathsf{leaf} & : \mathsf{T}_2 \\
\quad \mathsf{node} & : \mathsf{T}_2 \to \mathsf{T}_2 \to \mathsf{T}_2 \\
 &
\end{array}
\qquad
\begin{array}{ll}
\mathsf{data}\,\mathsf{R} & : \mathsf{T}_2 \to \mathsf{T}_2 \to \mathsf{Set} \\
\quad \mathsf{leaf}^\mathsf{R} & : \mathsf{R}\,\mathsf{leaf}\,\mathsf{leaf} \\
\quad \mathsf{node}^\mathsf{R} & : \mathsf{R}\,t\,t' \to \mathsf{R}\,s\,s' \to \mathsf{R}\,(\mathsf{node}\,t\,s)\,(\mathsf{node}\,t'\,s') \\
\quad \mathsf{perm}^\mathsf{R} & : \mathsf{R}\,(\mathsf{node}\,t\,t')\,(\mathsf{node}\,t'\,t)
\end{array}
$$

T_2 is the type of binary trees with no information at the nodes. R is an inductively defined binary relation on T_2. It has two congruence constructors for the two corresponding constructors of T_2 and a constructor perm^R expressing that exchanging two subtrees results in a related tree. Now we can define $\overline{\mathsf{T}}_2 := \mathsf{T}_2/\mathsf{R}$ and we lift the constructors leaf and node to $\overline{\mathsf{T}}_2$ as follows.

[2] A quotient type is just a quotient inductive type with the given constructors.

$$
\begin{aligned}
\overline{\mathsf{leaf}} &: \overline{\mathsf{T}}_2 := [\mathsf{leaf}] \\
\overline{\mathsf{node}} &: \overline{\mathsf{T}}_2 \to \overline{\mathsf{T}}_2 \to \overline{\mathsf{T}}_2 := \mathsf{RecT}_2/\mathsf{R}\,(\overline{\mathsf{T}}_2 \to \overline{\mathsf{T}}_2) \\
&\qquad\qquad \big(\lambda\,(t : \mathsf{T}_2).\mathsf{RecT}_2/\mathsf{R}\,\overline{\mathsf{T}}_2 \\
&\qquad\qquad\qquad \big(\lambda(t' : \mathsf{T}_2).[\mathsf{node}\,t\,t']\big) \\
&\qquad\qquad\qquad (\lambda p.[\mathsf{node}^{\mathsf{R}}\,\mathsf{refl}^{\mathsf{R}}\,p]{\equiv})\big) \\
&\qquad\qquad H
\end{aligned}
$$

In the above definition, $\mathsf{refl}^{\mathsf{R}}$ proves that R is reflexive and can be defined easily. The argument H needs to say that the previous argument respects the relation and up to functional extensionality and congruence it is given by $\lambda p.[\mathsf{node}^{\mathsf{R}}\,p\,\mathsf{refl}^{\mathsf{R}}]{\equiv}$. Note that $\overline{\mathsf{node}}$ works as expected: by the computation rule for quotients, we have that $\overline{\mathsf{node}}\,[t]\,[t'] = [\mathsf{node}\,t\,t']$.

Now we would like to do the same construction for infinitely branching trees which are given by the following datatype and relation.

$$
\begin{aligned}
&\mathsf{data}\,\mathsf{T} : \mathsf{Set} && \mathsf{data}\,\mathsf{Q} : \mathsf{T} \to \mathsf{T} \to \mathsf{Set} \\
&\quad \mathsf{leaf} : \mathsf{T} && \quad \mathsf{leaf}^{\mathsf{Q}} : \mathsf{Q}\,\mathsf{leaf}\,\mathsf{leaf} \\
&\quad \mathsf{node} : (\mathbb{N} \to \mathsf{T}) \to \mathsf{T} && \quad \mathsf{node}^{\mathsf{Q}} : \big(\forall n.\mathsf{Q}\,(f\,n)\,(g\,n)\big) \to \mathsf{Q}\,(\mathsf{node}\,f)\,(\mathsf{node}\,g) \\
& && \quad \mathsf{perm}^{\mathsf{Q}} : (f : \mathbb{N} \to \mathsf{T})(h : \mathbb{N} \to \mathbb{N}) \to \mathsf{isIso}\,h \\
& && \qquad \to \mathsf{Q}\,(\mathsf{node}\,f)\,\big(\mathsf{node}\,(f \circ h)\big)
\end{aligned}
$$

The branches are indexed by natural numbers and the permutation constructor says that we can precompose the function giving the subtrees with any function on natural numbers which is an isomorphism (isIso) and get a related tree.

We can lift leaf to the quotient T/Q just as in the binary case, however there seems to be no way doing this for the node constructor. In the binary case we had to nest the recursor for quotients twice and in the infinite case we would need to do this infinitely many times, but it is not clear how to express such an infinite definition in type theory.

However we can define $\overline{\mathsf{T}}$ in one step as a quotient inductive type and the problem disappears. We also don't need the congruence constructors for the relation anymore, as they can be proven by congruence of equality.

$$
\begin{aligned}
&\mathsf{data}\,\overline{\mathsf{T}}\ &&: \mathsf{Set}\\
&\quad \mathsf{leaf} &&: \overline{\mathsf{T}}\\
&\quad \mathsf{node} &&: (\mathbb{N} \to \overline{\mathsf{T}}) \to \overline{\mathsf{T}}\\
&\quad \mathsf{perm} &&: (f : \mathbb{N} \to \overline{\mathsf{T}})(h : \mathbb{N} \to \mathbb{N}) \to \mathsf{isIso}\, h \to \mathsf{node}\, f \equiv \mathsf{node}\,(f \circ h)
\end{aligned}
$$

For completeness, we write down the eliminator for this QIT.

$$
\begin{aligned}
&\mathsf{record}\,\mathsf{MElim}\overline{\mathsf{T}} : \mathsf{Set}_1\\
&\quad \overline{\mathsf{T}}^{\mathsf{M}} &&: \overline{\mathsf{T}} \to \mathsf{Set}\\
&\quad \mathsf{leaf}^{\mathsf{M}} &&: \overline{\mathsf{T}}^{\mathsf{M}}\,\mathsf{leaf}\\
&\quad \mathsf{node}^{\mathsf{M}} &&: \{f : \mathbb{N} \to \overline{\mathsf{T}}\}(f^M : (n : \mathbb{N}) \to \overline{\mathsf{T}}^{\mathsf{M}}\,(f\,n)) \to \overline{\mathsf{T}}^{\mathsf{M}}\,(\mathsf{node}\,f)\\
&\quad \mathsf{perm}^{\mathsf{M}} &&: \{f : \mathbb{N} \to \overline{\mathsf{T}}\}(f^M : (n : \mathbb{N}) \to \overline{\mathsf{T}}^{\mathsf{M}}\,(f\,n))(h : \mathbb{N} \to \mathbb{N})\\
& &&\qquad (p : \mathsf{isIso}\,h) \to \mathsf{node}^{\mathsf{M}}\,f^M \equiv^{\mathsf{perm}\,f\,h\,p} \mathsf{node}^{\mathsf{M}}\,(f^M \circ h)
\end{aligned}
$$

$$
\begin{aligned}
&\mathsf{module}\,\mathsf{Elim}\overline{\mathsf{T}}\,(m : \mathsf{MElim}\overline{\mathsf{T}})\\
&\quad \mathsf{open}\,\mathsf{MElim}\overline{\mathsf{T}}\,m\\
&\quad \mathsf{Elim}\overline{\mathsf{T}} : (t : \overline{T}) &&\to \overline{\mathsf{T}}^{\mathsf{M}}\,t\\
&\quad \mathsf{Elim}\overline{\mathsf{T}}\ \ \mathsf{leaf} &&= \mathsf{leaf}^{\mathsf{M}}\\
&\quad \mathsf{Elim}\overline{\mathsf{T}}\ \ (\mathsf{node}\,f) &&= \mathsf{node}^{\mathsf{M}}\,\big(\lambda n.\mathsf{Elim}\overline{\mathsf{T}}\,(f\,n)\big)
\end{aligned}
$$

Quotient inductive types are a convenient way to define generalised algebraic theories [36]. The motive and the methods for the recursor collected together into a record form an algebra for the theory defined by the QIT. This algebra can be viewed as a model of the theory where soundness is ensured by the methods for the equality constructors. The syntax can be viewed as the initial (term) model and initiality is given by the recursor and its computation rules.

QITs are special cases of HITs which is a very active research area. Higher inductive types are proposed in chapter 6 of the book on homotopy type theory [88] but only examples are given without a general definition. There are a few proposals for describing what HITs are, but none of them cover the most general case. Sojakova describes a special case called W-suspensions and proves that these have an initiality property in a certain category [92]. Henning et al [22] define a syntactic scheme for HITs with only 1-constructors. Codes for certain mutual and higher inductive types are given by Capriotti [34]. Some HITs can be reduced to HITs without recursive higher constructors [64]. If this is the case in general, it might lead to a simpler description of HITs. Lumsdaine and Shulman study the semantics of HITs [67].

In chapter 3 we will use the combination of inductive inductive types and quotient inductive types that we call quotient inductive inductive types (QIITs). The syntactic description of QIITs is ongoing research, see the upcoming thesis of Gabe Dijkstra [47]. Here we give an informal description of what a QIIT is. A QIIT is given first as a list of sorts, and then a list of constructors. The sorts can depend on each other, e.g. $\mathsf{A} : \mathsf{Set}$, $\mathsf{B} : \mathsf{A} \to \mathsf{Set}$, $\mathsf{C} : (a : \mathsf{A}) \to \mathsf{B}\, a \to \mathsf{Set}$. The codomain of each sort has to be Set. Note that the dependencies between sorts can be quite intricate, e.g. B might have sort $((\mathsf{A} \to \mathsf{A}) \to \mathsf{A}) \to \mathsf{Set}$ or $(a : \mathsf{A}) \to a \equiv a \to \mathsf{Set}$. The codomain of constructors need to be one of the sorts or an equality between elements in a sort. For example, we can have $\mathsf{a} : \mathsf{A}$ or $\mathsf{p} : (x : \mathsf{A}) \to x \equiv \mathsf{a}$. Constructors can refer to previously defined constructors, for example we might have $\mathsf{b} : \mathsf{B}\, \mathsf{a}$. Constructors need to obey strict positivity, e.g. $\mathsf{a}' : (\mathsf{A} \to \mathsf{A}) \to \mathsf{A}$ is not allowed because A has a negative occurrence in the type of the first and only argument of the constructor. The constructor $\mathsf{b} : (\mathsf{A} \to \mathsf{A}) \to \mathsf{B}\, \mathsf{a}$ however is allowed as the negative occurrence is for A and not B which is the target of the constructor. Constructors can refer to equalities as well and probably these also need to obey the strict positivity condition, for example, a constructor $\mathsf{p}' : ((a \equiv a) \to A) \to A$ is questionable.

In section 4.3 we will describe a general method for deriving the eliminator of a closed QIIT. There we simply define QIITs as contexts, for example a context $A : \mathsf{Set}, B : A \to \mathsf{Set}, a : A, b : B\, a$ describes a QIIT with two sorts and two constructors. The variable names are the names for the sorts and constructors. This description makes sure that later sorts and constructors can only refer to previous ones, but strict positivity is not enforced. The motives and methods for an eliminator can be however extracted for such a context in general.

We hope that, after giving a definition of QIITs, they can be justified using a setoid model [8].

Adding QIITs (and functional extensionality, which is also a consequence of the existence of the interval QIT, see formalisation) to the theory poses a canonicity problem, i.e. can all closed terms of type ℕ be reduced to numerals. Observational type theory [19] is a strict type theory tackling this problem and cubical type theory [40] is a non strict type theory giving computational explanation of homotopy type theory. These theories don't yet have reliable implementations and their metatheory is not yet developed, this is why we have chosen the more conservative way of working in a strict theory with axioms.

QIITs can be added to Agda by postulating all the constructors and adding the computation rules as rewrite rules [39]. This avoids the problems with previous approaches such as [66] using inductive types and only postulating the equality constructors.

Chapter 3

Syntax and standard semantics of type theory

In this chapter we explain how we arrive at our definition of syntax starting from an informal presentation of a basic type theory, then we define the syntax and its elimination principle. Our syntax includes dependent function space, a base type and a family of types over the base type. We show how to reason using this syntax and show how to use the elimination principle to define functions from the syntax to another type. More specifically, we prove disjointness of the empty context and extended contexts, we prove injectivity of the type constructor Π, and define the standard interpretation of type theory in which every object theoretic construct is modelled by its metatheoretic counterpart.

3.1 From informal to formal syntax

Type theory is a formal system for deriving judgements: it consists of a collection of derivation rules each of which has one judgement as a conclusion. Figure 3.1 lists the derivation rules of a basic type theory with universes à la Russel and Π types. The notation used here is close to how one informally writes type theory on a whiteboard.

We have three kinds of judgements: (1) Γ is a valid context, denoted $\Gamma \vdash$, (2) term t has type A in context Γ, denoted $\Gamma \vdash t : A$, and (3) two terms are convertible, denoted $\Gamma \vdash t \sim u : A$.

The derivation rules come in groups: there are a few core rules, and then for

(1) Contexts:

$$\frac{}{\cdot \vdash}\ \text{C-empty} \qquad \frac{\Gamma \vdash \qquad \Gamma \vdash A : \mathsf{U}}{\Gamma, x : A \vdash}\ \text{C-ext}$$

(2) Terms:

$$\frac{\Gamma \vdash A : \mathsf{U}}{\Gamma, x : A \vdash x : A}\ \text{var} \qquad \frac{\Gamma \vdash t : A \qquad \Gamma \vdash B : \mathsf{U}}{\Gamma, x : B \vdash t : A}\ \text{wk} \qquad \frac{\Gamma \vdash}{\Gamma \vdash \mathsf{U} : \mathsf{U}}\ \mathsf{U}\text{-I}$$

$$\frac{\Gamma \vdash A : \mathsf{U} \qquad \Gamma, x : A \vdash B : \mathsf{U}}{\Gamma \vdash \Pi(x : A).B : \mathsf{U}}\ \Pi\text{-F} \qquad \frac{\Gamma, x : A \vdash t : B}{\Gamma \vdash \lambda x.t : \Pi(x : A).B}\ \Pi\text{-I}$$

$$\frac{\Gamma \vdash f : \Pi(x : A).B \qquad \Gamma \vdash a : A}{\Gamma \vdash f\,a : B[x := a]}\ \Pi\text{-E}$$

$$\frac{\Gamma \vdash A \sim B : \mathsf{U} \quad \Gamma \vdash t : A}{\Gamma \vdash t : B}\ \text{t-coe}$$

(3) Conversion for terms:

$$\frac{\Gamma \vdash t : A}{\Gamma \vdash t \sim t : A}\ \text{t-eq-refl} \qquad \frac{\Gamma \vdash u \sim v : A}{\Gamma \vdash v \sim u : A}\ \text{t-eq-sym}$$

$$\frac{\Gamma \vdash u \sim v : A \quad \Gamma \vdash v \sim w : A}{\Gamma \vdash u \sim w : A}\ \text{t-eq-trans}$$

$$\frac{\Gamma \vdash A \sim B : \mathsf{U} \quad \Gamma \vdash u \sim v : A}{\Gamma \vdash u \sim v : B}\ \text{t-eq-coe}$$

$$\frac{\Gamma \vdash A \sim A' : \mathsf{U} \qquad \Gamma, x : A \vdash B \sim B' : \mathsf{U}}{\Gamma \vdash \Pi(x : A).B \sim \Pi(x : A').B' : \mathsf{U}}\ \Pi\text{-F-cong}$$

$$\frac{\Gamma, x : A \vdash t \sim t' : B}{\Gamma \vdash \lambda x.t \sim \lambda x.t' : \Pi(x : A).B}\ \Pi\text{-I-cong}$$

$$\frac{\Gamma \vdash f \sim f' : \Pi(x : A).B \qquad \Gamma \vdash a \sim a' : A}{\Gamma \vdash f\,a \sim f'\,a' : B[x := a]}\ \Pi\text{-E-cong}$$

$$\frac{\Gamma, x : A \vdash t : B}{\Gamma \vdash (\lambda x.t)\,a \sim t[x := a] : B[x := a]}\ \Pi\text{-}\beta \qquad \frac{\Gamma \vdash f : \Pi(x : A).B}{\Gamma \vdash f \sim (\lambda x.f\,x) : \Pi(x : A).B}\ \Pi\text{-}\eta$$

Figure 3.1: Informal derivation rules for a basic type theory grouped by the kind of judgement. Rules for α-conversion, universe indices and the definition of substitution are omitted.

Example A: the polymorphic identity function.

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\cdot\vdash}\ \text{C-empty}}{\cdot\vdash\mathsf{U}:\mathsf{U}}\ \mathsf{U}\text{-I}}{\cdot,A:\mathsf{U}\vdash A:\mathsf{U}}\ \text{var}}{\cdot,A:\mathsf{U},x:A\vdash x:A}\ \text{var}}{\cdot,A:\mathsf{U}\vdash\lambda x.x:\Pi(x:A).A}\ \Pi\text{-I}}{\cdot\vdash\lambda A.\lambda x.x:\Pi(A:\mathsf{U}).\Pi(x:A).A}\ \Pi\text{-I}$$

Example B: predicate space on a type variable.

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\cdot\vdash}\ \text{C-empty}}{\cdot\vdash\mathsf{U}:\mathsf{U}}\ \mathsf{U}\text{-I}}{\cdot,A:\mathsf{U}\vdash A:\mathsf{U}}\ \text{var}\qquad\dfrac{\dfrac{\dfrac{\dfrac{}{\cdot\vdash}\ \text{C-empty}\quad\dfrac{\dfrac{}{\cdot\vdash}\ \text{C-empty}}{\cdot\vdash\mathsf{U}:\mathsf{U}}\ \mathsf{U}\text{-I}}{\cdot,A:\mathsf{U}\vdash}\ \text{C-ext}\qquad\dfrac{\dfrac{\dfrac{}{\cdot\vdash}\ \text{C-empty}}{\cdot\vdash\mathsf{U}:\mathsf{U}}\ \mathsf{U}\text{-I}}{\cdot,A:\mathsf{U}\vdash A:\mathsf{U}}\ \text{var}}{\cdot,A:\mathsf{U},x:A\vdash}\ \text{C-ext}}{\cdot,A:\mathsf{U},x:A\vdash\mathsf{U}:\mathsf{U}}\ \mathsf{U}\text{-I}}{\cdot,A:\mathsf{U}\vdash\Pi(x:A).\mathsf{U}:\mathsf{U}}\ \Pi\text{-F}$$

Example C: deriving the natural number 2 in a context representing natural numbers. We abbreviate the following derivation by p.

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\cdot\vdash}\ \text{C-empty}}{\cdot\vdash\mathsf{U}:\mathsf{U}}\ \mathsf{U}\text{-I}}{\cdot,N:\mathsf{U}\vdash N:\mathsf{U}}\ \text{var}\qquad\dfrac{\dfrac{\dfrac{}{\cdot\vdash}\ \text{C-empty}}{\cdot\vdash\mathsf{U}:\mathsf{U}}\ \mathsf{U}\text{-I}}{\cdot,N:\mathsf{U}\vdash N:\mathsf{U}}\ \text{var}}{\cdot,N:\mathsf{U},z:N\vdash N:\mathsf{U}}\ \text{wk}$$

We abbreviate the following derivation by q.

$$\dfrac{p\qquad\dfrac{p\quad p}{\cdot,N:\mathsf{U},z:N,x:N\vdash N:\mathsf{U}}\ \text{wk}}{\cdot,N:\mathsf{U},z:N\vdash\Pi(x:N).N:\mathsf{U}}\ \Pi\text{-F}$$

We abbreviate the context $\cdot,N:\mathsf{U},z:\mathbb{N},s:(x:N),N$ by Γ.

$$\dfrac{\dfrac{q}{\Gamma\vdash s:\Pi(x:N).N}\ \text{var}\qquad\dfrac{\dfrac{q}{\Gamma\vdash s:\Pi(x:N).N}\ \text{var}\qquad\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\cdot\vdash}\ \text{C-empty}}{\cdot\vdash\mathsf{U}:\mathsf{U}}\ \mathsf{U}\text{-F}}{\cdot,N:\mathsf{U}\vdash N:\mathsf{U}}\ \text{var}}{\cdot,N:\mathsf{U},z:N\vdash z:N}\ \text{var}\qquad q}{\Gamma\vdash z:N}\ \text{wk}}{\Gamma\vdash s\,z:N}\ \Pi\text{-E}}{\Gamma\vdash s\,(s\,z):N}\ \Pi\text{-E}$$

Figure 3.2: Example derivation trees using the rules of figure 3.1.

each type former there is a separate group of rules. In our basic type theory we have two type formers, U and Π. The rules belonging to the group of U and Π have names starting with U and Π, respectively. The rules for each type former follow the same structure: type formation rule (denoted -F), introduction rule (-I), elimination rule (-E), computation rule (-β) and uniqueness rule (-η). The conversion rules state that conversion is an equivalence relation, and we have coercion rules (convertible objects are interchangeable) and congruence rules (every construction respects the conversion relation).

We haven't listed the rules which say that terms are identified up to α-conversion and that name capture is avoided by appropriate renaming. We also refrained from writing down the definition of substitution. We also assume that we have a hierarchy of universes, i.e. $\mathsf{U}_i : \mathsf{U}_{i+1}$, but informally we refrain from writing the indices.

We say that a judgement can be derived in type theory when there is a derivation tree ending with that judgement. We give an example of a few derivation trees in figure 3.2.

Traditionally, the syntax of type theory is formalised starting with an infinite set of variable names and an alphabet which includes this set and symbols such as λ, Π etc. Then one defines the inductive sets of preterms, pretypes, precontexts. Finally, a ternary typing relation is defined as a subset of the cartesian product of precontexts, pretypes and preterms. We can view our presentation in figure 3.1 as an inductive definition of such a relation (leaving the low-level details such as the definition of preterms implicit). Well typed terms are given by the preterms for which there is a pretype and a precontext such that this relation holds. For examples of such presentations, see section 2.3 in [58] or chapter 4 in [94]. We view this approach as a low-level one, because whenever we would like to talk about terms or derivations we always need to go back and deal with lists of symbols.

When using type theory as a metalanguage we are able to define terms at a higher level, even formally. We will define the type of terms as a family of types indexed over contexts and types.

$$\mathsf{Tm} : (\Gamma : \mathsf{Con}) \to \mathsf{Ty}\,\Gamma \to \mathsf{Set}$$

This way we don't have preterms but only well typed terms which contain all the information for reproducing the derivation tree. Terms and derivations are identified: e.g. the rule Π-I and λ will be identified and given the name lam. Similarly we will only formalise well-scoped types and valid contexts. This approach is sometimes called intrinsically typed syntax. An advantage of intrinsically typed syntax is that we don't need coherence theorems between operations defined on the syntax and typing: e.g. type preservation comes for free because everything is well-typed; similarly, we don't need the substitution lemma as substitution is not defined on preterms but is defined on well typed terms using a typing rule.

Rules Π-E and Π-β mention substitutions. In the traditional presentation separating preterms and well-typed terms, substitution is defined by recursion on preterms and it can be proven that substitution respects the typing relation. In an intrinsically typed syntax, it is also possible to define substitution recursively. This results in an inductive inductive recursive definition (the notion of induction induction recursion is studied in section 6.1 of Forsberg's thesis [79]). An example of such a definition of type theory is Danielsson's work [43]. An alternative to defining substitution recursively is to build substitutions into the syntax. This is called explicit substitution calculus and was originally introduced to bridge the gap between λ-calculus and its implementations [1]. In this approach substitution is a separate term former (constructor) which builds a new term from a term and a substitution. In fact, Danielsson [43] uses a mixed approach: he defines substitutions on types implicitly (recursively) and substitution on terms explicitly. He says that originally he wanted to define a fully implicit syntax, but when defining implicit substitutions, one needs to prove substitution lemmas mutually with the syntax and it is hard to show that for such a complicated definition, every function terminates.

We take the conceptually cleaner approach of fully explicit substitutions. We use parallel substitutions, that is, a substitution is a lists of terms. Its domain is the context in which all the terms are defined and its codomain is the context which is built up from the types of the terms. This also provides a natural way to deal with variable names and α-conversion. We use De Bruijn indices [33] encoded by projection substitutions: there will be a projection π_1 which forgets the last component of a substitution, and a projection π_2 which projects out the last term from a substitution.

As terms are indexed over types, we need to declare them separately, first types and then terms. Hence, we can't just say that a type is an element of the universe as in the informal presentation of figure 3.1. We need to have universes à la Tarski instead of à la Russel.

The notation of figure 3.1 is very concise in that, for example, the rule Π-I does not say that $\Gamma \vdash A : \mathsf{U}$ and $\Gamma, x : A \vdash B : \mathsf{U}$ (and indeed, this assumption is not necessary – it is admissible). In our syntax however we work in type theory, hence we need to list every argument with its type. However, we can still achieve the same conciseness in the formal presentation by adding these additional assumptions as implicit arguments to the syntax. For example, we write the type of the constructor lam as follows.

$$\mathsf{lam} : \mathsf{Tm}\,(\Gamma, A)\,B \to \mathsf{Tm}\,\Gamma\,(\Pi\,A\,B)$$

Spelling out the implicit arguments, this means the following.

$$\mathsf{lam} : \{\Gamma : \mathsf{Con}\}\{A : \mathsf{Ty}\,\Gamma\}\{B : \mathsf{Ty}\,(\Gamma, A)\} \to \mathsf{Tm}\,(\Gamma, A)\,B \to \mathsf{Tm}\,\Gamma\,(\Pi\,A\,B)$$

This way our syntax matches the usual informal notation of type theory. Without implicit arguments we would have to write $\mathsf{lam}\,\Gamma\,A\,B\,t$ instead of $\mathsf{lam}\,t$.

Formalising the type theory of figure 3.1 following the above considerations would result in an inductive inductive type with the following eight constituent types for contexts, types, substitutions (Tms for lists of terms), terms and conversion for all of them in the same order.

$$\begin{aligned}
&\mathsf{data}\,\mathsf{Con} && : \mathsf{Set}\\
&\mathsf{data}\,\mathsf{Ty} && : \mathsf{Con} \to \mathsf{Set}\\
&\mathsf{data}\,\mathsf{Tms} && : \mathsf{Con} \to \mathsf{Con} \to \mathsf{Set}\\
&\mathsf{data}\,\mathsf{Tm} && : (\Gamma : \mathsf{Con}) \to \mathsf{Ty}\,\Gamma \to \mathsf{Set}\\
&\mathsf{data} - \sim_{\mathsf{Con}} - && : \mathsf{Con} \to \mathsf{Con} \to \mathsf{Set}\\
&\mathsf{data} - \sim_{\mathsf{Ty}} - && : \mathsf{Ty}\,\Gamma \to \mathsf{Ty}\,\Gamma \to \mathsf{Set}\\
&\mathsf{data} - \sim_{\mathsf{Tms}} - && : \mathsf{Tms}\,\Gamma\,\Delta \to \mathsf{Tms}\,\Gamma\,\Delta \to \mathsf{Set}\\
&\mathsf{data} - \sim_{\mathsf{Tm}} - && : \mathsf{Tm}\,\Gamma\,A \to \mathsf{Tm}\,\Gamma\,A \to \mathsf{Set}
\end{aligned}$$

We cannot avoid defining conversion at the same time as the first four types because of the rule t-coe which becomes a constructor of Tm mentioning the conversion relation. In contrast, for simple type theory there is no such rule and the conversion relation can be given in a separate step after defining the syntax.

However, defining type theory with these eight constituent types results in lots of boilerplate because of the conversion rules. Most of these rules are so obvious that introductions to type theory do not even list them explicitly (e.g. [58], [88]). Basically, one needs to work with setoids instead of types and then families of setoids indexed over setoids. (A setoid is a type together with an equivalence relation which we view as the equality of that type [8], e.g. we only define functions between setoids which respect the corresponding relations.) The amount of type theoretic boilerplate makes using such a definition infeasible in practice. An example is Chapman's work [38] which managed to give a complete presentation using such an approach.

If we look at the conversion rules, we observe that there is another relation in type theory for which reflexivity, symmetry, transitivity, coercion and the congruence rules hold: equality (section 2.2.2). The question naturally arises whether we can use equality as a tool for abstracting over these properties. Note that we also need to account for the interesting conversion rules, namely Π-β and Π-η. Quotient inductive types provide a solution precisely for this: they allow the addition of equality constructors for the interesting cases and all the other rules become automatically true by the corresponding rules for equality.

In addition, the syntax given as a QIIT is a more abstract definition than the one with explicit conversion relations. The quotienting enforces that we cannot distinguish between convertible syntactic objects.

3.2 The syntax

We define the syntax of type theory as a quotient inductive inductive type consisting of four different types: contexts, types, substitutions and terms.

$$\begin{aligned}
&\mathsf{data}\,\mathsf{Con} &&: \mathsf{Set}\\
&\mathsf{data}\,\mathsf{Ty} &&: \mathsf{Con} \to \mathsf{Set}\\
&\mathsf{data}\,\mathsf{Tms} &&: \mathsf{Con} \to \mathsf{Con} \to \mathsf{Set}\\
&\mathsf{data}\,\mathsf{Tm} &&: (\Gamma : \mathsf{Con}) \to \mathsf{Ty}\,\Gamma \to \mathsf{Set}
\end{aligned}$$

Contexts are given by a type. Types are indexed over contexts. As we have dependent types, a type is only valid in a given context. Substitutions are indexed over two contexts. One can think of $\mathsf{Tms}\,\Gamma\,\Delta$ as a sequence of terms in context Γ which inhabit all types in Δ, hence the name Tms. Terms are indexed over a context and a type in that context.

We will give the constructors for the four constituent types in separate groups. First we spell out the core substitution calculus, and then we add rules for different type formers separately.

3.2.1 The core type theory

The constructors of the substitution calculus are given below. The substitution calculus does not include any type formers such as Π. We will extend the QIIT of the syntax with those later.

$$
\begin{array}{lll}
\multicolumn{3}{l}{\mathsf{data\,Con}} \\
& \cdot & : \mathsf{Con} \\
& -,- & : (\Gamma : \mathsf{Con}) \to \mathsf{Ty}\,\Gamma \to \mathsf{Con} \\
\multicolumn{3}{l}{\mathsf{data\,Ty}} \\
& -[-] & : \mathsf{Ty}\,\Theta \to \mathsf{Tms}\,\Gamma\,\Theta \to \mathsf{Ty}\,\Gamma \\
\multicolumn{3}{l}{\mathsf{data\,Tms}} \\
& -\circ- & : \mathsf{Tms}\,\Theta\,\Delta \to \mathsf{Tms}\,\Gamma\,\Theta \to \mathsf{Tms}\,\Gamma\,\Delta \\
& \mathsf{id} & : \mathsf{Tms}\,\Gamma\,\Gamma \\
& \epsilon & : \mathsf{Tms}\,\Gamma\,\cdot \\
& -,- & : (\sigma : \mathsf{Tms}\,\Gamma\,\Delta) \to \mathsf{Tm}\,\Gamma\,A[\sigma] \to \mathsf{Tms}\,\Gamma\,(\Delta, A) \\
& \pi_1 & : \mathsf{Tms}\,\Gamma\,(\Delta, A) \to \mathsf{Tms}\,\Gamma\,\Delta \\
\multicolumn{3}{l}{\mathsf{data\,Tm}} \\
& -[-] & : \mathsf{Tm}\,\Theta\,A \to (\sigma : \mathsf{Tms}\,\Gamma\,\Theta) \to \mathsf{Tm}\,\Gamma\,A[\sigma] \\
& \pi_2 & : (\sigma : \mathsf{Tms}\,\Gamma\,(\Delta, A)) \to \mathsf{Tm}\,\Gamma\,A[\pi_1\,\sigma] \\
\multicolumn{3}{l}{\mathsf{data\,Ty}} \\
& [][] & : A[\sigma][\nu] \equiv A[\sigma \circ \nu] \\
& [\mathsf{id}] & : A[\mathsf{id}] \equiv A \\
\multicolumn{3}{l}{\mathsf{data\,Tms}} \\
& \circ\circ & : (\sigma \circ \nu) \circ \delta \equiv \sigma \circ (\nu \circ \delta) \\
& \mathsf{id}\circ & : \mathsf{id} \circ \sigma \equiv \sigma \\
& \circ\mathsf{id} & : \sigma \circ \mathsf{id} \equiv \sigma \\
& \epsilon\eta & : \{\sigma : \mathsf{Tms}\,\Gamma\,\cdot\} \to \sigma \equiv \epsilon
\end{array}
$$

$$\pi_1\beta \quad : \pi_1\,(\sigma, t) \equiv \sigma$$

$$\pi\eta \quad : (\pi_1\,\sigma, \pi_2\,\sigma) \equiv \sigma$$

$$,\circ \quad : (\nu, t) \circ \sigma \equiv (\nu \circ \sigma), ({}_{[][]*}t[\sigma])$$

$\mathsf{data\,Tm}$

$$\pi_2\beta \quad : \pi_2\,(\sigma, t) \equiv^{\pi_1\beta} t$$

We explain the above definition in detail now.

There are two ways to construct a context: the empty context $\cdot$ and context extension $-,-$ which adds a type in the context which we extend.

We are able to substitute types and terms and substitutions themselves (the latter is called composition). That is, given a type $A : \mathsf{Ty}\,\Theta$, term $t : \mathsf{Tm}\,\Theta\,A$ and a substitution $\nu : \mathsf{Tms}\,\Theta\,\Delta$, all interpreted in context Θ, we can interpret them in Γ by a substitution $\sigma : \mathsf{Tms}\,\Gamma\,\Theta$ which interprets all of Θ in Γ. This is expressed by $A[\sigma] : \mathsf{Ty}\,\Gamma$, $t[\sigma] : \mathsf{Tm}\,\Gamma\,A[\sigma]$ and $\nu \circ \sigma : \mathsf{Tms}\,\Gamma\,\Delta$. Substitution is associative which is expressed by $[][]$ for types and $\circ\circ$ for substitutions. The similar rule for terms can be derived, see section 3.3.

We have the identity substitution id which is identity when we substitute types ($[\mathsf{id}]$) and substitutions ($\mathsf{id}\circ$, $\circ\mathsf{id}$). Types can be only substituted from one side, this is why we only have one identity law for them. Again, the similar rule for terms can be derived.

The constructor ϵ provides the empty substitution and $\epsilon\eta$ witnesses that every substitution into the empty context is equal to it.

The substitution extension operator $-,-$ is adding a term at the end of a substitution: given a σ providing all types in Δ, we need a term of type $A[\sigma]$ to construct a substitution into Δ, A. We have the two projections π_1 and π_2 which forget and project out the last term, respectively. They work as expected: projection after $-,-$ gives the expected results (rules $\pi_1\beta$ and $\pi_2\beta$). We have the other direction as well: if we project out the first and second components and then join them by substitution extension, we get the original substitution ($\pi\eta$). Finally, we have the law $,\circ$ which tells us what happens if we substitute an extended substitution: the substitution just goes under the $-,-$ in a pointwise fashion.

Note that in the definition of $,\circ$ the term $t[\sigma]$ has type $A[\nu][\sigma]$ but $(\nu \circ \sigma), -$ requires a term of type $A[\nu \circ \sigma]$. This is why we need to transport $t[\sigma]$ along the equality $[][]$. Similarly when expressing the type of $\pi_2\beta$ we have a term of type on the left hand side $A[\pi_1\,(\sigma, t)]$ and a term of type $A[\sigma]$ on the right hand side. These types are equal by $\pi_1\beta$, hence $\pi_2\beta$ is an equality over $\pi_1\beta$.

We can summarize the syntax as follows.

- Contexts and substitutions form a category with a terminal object.

- Types form a family of types over contexts, terms form a family over contexts and types and they are functorial.

- We have a substitution extension operation given by the following natural isomorphism.

$$\pi_1\beta, \pi_2\beta \curvearrowbotleft \qquad -,- \downarrow \; \frac{\sigma : \mathsf{Tms}\,\Gamma\,\Delta \qquad \mathsf{Tm}\,\Gamma\,A[\sigma]}{\mathsf{Tms}\,\Gamma\,(\Delta, A)} \uparrow \pi_1, \pi_2 \qquad \curvearrowright \pi\eta$$

  Naturality is expressed by $,\circ$. If one direction of an isomorphism is natural, so is the other, this is why we only state this direction. We prove the other direction (given by $\pi_1\circ$ and $\pi_2[]$) later.

Using our categorically inspired syntax we can derive more traditional syntactic constructions such as variables expressed as typed De Bruijn indices [33]. For this we first define the weakening substitution.

$$\mathsf{wk} : \mathsf{Tms}\,(\Gamma, A)\,\Gamma := \pi_1\,\mathsf{id}$$

When we project out the last element from the context, we need to weaken the type because now we are in an extended context. The successor constructor for De Bruijn variables is just substitution by weakening.

$$\begin{aligned} &\mathsf{vz} && : \mathsf{Tm}\,(\Gamma, A)\,(A[\mathsf{wk}]) := \pi_2\,\mathsf{id} \\ &\mathsf{vs}\,(x : \mathsf{Tm}\,\Gamma\,A) && : \mathsf{Tm}\,(\Gamma, B)\,(A[\mathsf{wk}]) := x[\mathsf{wk}] \end{aligned}$$

3.2.2 A base type and family

We add a base type U and a family El over this type. El is the only place in our basic syntax where terms leak into types. U and El are the dependently typed analogs of the base type ι in simple type theory: they are the simplest type formers which make the theory nontrivial.

We add the type formation rules and the substitution rules. We write the symbol + after Ty to show that these are additional constructors to the ones given in the previous section.

$$\begin{array}{ll}
\multicolumn{2}{l}{\mathsf{data}\,\mathsf{Ty}+} \\
\quad \mathsf{U} & : \mathsf{Ty}\,\Gamma \\
\quad \mathsf{El} & : \mathsf{Tm}\,\Gamma\,\mathsf{U} \to \mathsf{Ty}\,\Gamma \\
\quad \mathsf{U}[] & : \mathsf{U}[\sigma] \equiv \mathsf{U} \\
\quad \mathsf{El}[] & : (\mathsf{El}\,\hat{A})[\sigma] \equiv \mathsf{El}\,({}_{\mathsf{U}[]*}\hat{A}[\sigma])
\end{array}$$

The type constructor U is not affected by substitutions and substituting El pushes the substitution σ through to the term of type U which gains type $\mathsf{U}[\sigma]$, this is why we need to transport along $\mathsf{U}[]$.

3.2.3 Dependent function space

First we define lifting of a substitution: the operation $\uparrow$ takes a substitution σ from Γ to Δ and returns an extended substitution from $\Gamma, A[\sigma]$ to Δ, A which does not touch the last element in the context. It works by weakening σ and extending the substitution by the referring to the last element in the context.

$$(\sigma : \mathsf{Tms}\,\Gamma\,\Delta) \uparrow A : \mathsf{Tms}\,(\Gamma, A[\sigma])\,(\Delta, A) := (\sigma \circ \mathsf{wk}), ({}_{[][]*}\mathsf{vz})$$

Now we define function space by the type formation rule, abstraction, application, the β computation rule, the η uniqueness rule and substitution rules for type formation and lambda.

$$\begin{array}{ll}
\multicolumn{2}{l}{\mathsf{data}\,\mathsf{Ty}+} \\
\quad \Pi & : (A : \mathsf{Ty}\,\Gamma) \to \mathsf{Ty}\,(\Gamma, A) \to \mathsf{Ty}\,\Gamma \\
\quad \Pi[] & : (\Pi\,A\,B)[\sigma] \equiv \Pi\,(A[\sigma])\,(B[\sigma \uparrow A]) \\
\multicolumn{2}{l}{\mathsf{data}\,\mathsf{Tm}+} \\
\quad \mathsf{lam} & : \mathsf{Tm}\,(\Gamma, A)\,B \to \mathsf{Tm}\,\Gamma\,(\Pi\,A\,B) \\
\quad \mathsf{app} & : \mathsf{Tm}\,\Gamma\,(\Pi\,A\,B) \to \mathsf{Tm}\,(\Gamma, A)\,B \\
\quad \Pi\beta & : \mathsf{app}\,(\mathsf{lam}\,t) \equiv t \\
\quad \Pi\eta & : \mathsf{lam}\,(\mathsf{app}\,t) \equiv t \\
\quad \mathsf{lam}[] & : (\mathsf{lam}\,t)[\sigma] \equiv^{\Pi[]} \mathsf{lam}\,(t[\sigma \uparrow A])
\end{array}$$

This definition can be summarized as a natural isomorphism where naturality is given by $\mathsf{lam}[]$.

$$\Pi\beta \circlearrowleft \qquad \mathsf{lam}\downarrow \frac{\mathsf{Tm}\,(\Gamma, A)\,B}{\mathsf{Tm}\,\Gamma\,(\Pi\,A\,B)} \uparrow \mathsf{app} \qquad \curvearrowright \Pi\eta$$

We use the categorical app operator, but the standard one $(-\$-)$ can also be derived.

$$\begin{aligned}
&\langle (u : \mathsf{Tm}\,\Gamma\,A)\rangle && : \mathsf{Tms}\,\Gamma\,(\Gamma, A) := \mathsf{id}, {}_{[\mathsf{id}]^{-1}*}u \\
&(t : \mathsf{Tm}\,\Gamma\,(\Pi\,A\,B))\$(u : \mathsf{Tm}\,\Gamma\,A) : B[\langle u\rangle] && := (\mathsf{app}\,t)[\langle u\rangle]
\end{aligned}$$

The substitution $\langle u\rangle$ is identity on the first part of the context and provides u for the last type.

The substitution law for app can be derived using $\mathsf{lam}[]$ and the β and η rules.

$$\begin{aligned}
\mathsf{app}[] \ : \ & (\mathsf{app}\,t)[\sigma \uparrow A] \\
& && (\Pi\beta^{-1}) \\
& \equiv \mathsf{app}\,\Big(\mathsf{lam}\big((\mathsf{app}\,t)[\sigma \uparrow A]\big)\Big) \\
& && (\mathsf{lam}[]^{-1}) \\
& \equiv \mathsf{app}\,\big({}_{\Pi[]*}\mathsf{lam}\,(\mathsf{app}\,t)[\sigma]\big) \\
& && (\Pi\eta) \\
& \equiv \mathsf{app}\,(t[\sigma])
\end{aligned}$$

3.3 Reasoning in the syntax

In this section we show how to prove theorems using the syntax through a couple of examples, including those of figure 3.2. We also show how we used heterogeneous equality in the formalisation to make reasoning easier. Apart from this section, when writing equality reasoning this thesis uses extensional type theory for readability.

First we prove the substitution laws for π_1 and π_2. The one for π_1 is given by the following equational reasoning.

$$
\begin{aligned}
\pi_1{\circ} \; : \;\; & \pi_1\,\nu \circ \sigma \\
& \qquad\qquad\qquad\qquad (\pi_1\beta^{-1}) \\
& \equiv \pi_1\,\big(\pi_1\,\nu \circ \sigma, (\pi_2\,\nu)[\sigma]\big) \\
& \qquad\qquad\qquad\qquad ({,}{\circ}^{-1}) \\
& \equiv \pi_1\,\big((\pi_1\,\nu, \pi_2\,\nu) \circ \sigma\big) \\
& \qquad\qquad\qquad\qquad (\pi\eta) \\
& \equiv \pi_1\,(\nu \circ \sigma)
\end{aligned}
$$

We need $\pi_1\circ$ even to state the substitution law for π_2.

$$\pi_2[] : (\pi_2\,\nu)[\sigma] \equiv^{[][] \centerdot \mathsf{ap}\,(A[-])\,\pi_1\circ} \pi_2\,(\nu \circ \sigma)$$

The left hand side has type $\mathsf{Tm}\,\Gamma\,(A[\pi_1\,\nu][\sigma])$ and the right hand side has type $\mathsf{Tm}\,\Gamma\,(A[\pi_1\,(\nu \circ \sigma)])$. Hence we need to transport the left hand side through the equality $[][] \centerdot \mathsf{ap}\,(A[-])\,\pi_1\circ$.

To ease working with such equalities we introduce heterogeneous equality [19]. We will only use heterogeneous equality in this section to illustrate how we formalised the constructions in this thesis (which use the notation of extensional type theory).

A heterogeneous equality between two elements of two different types is a pair of an equality between the two types and an equality of the elements up to this equality.

$$(a : A) \simeq (b : B) := \Sigma(p : A \equiv B).a \equiv^p b$$

An equivalent form of this is relation is called John Major equality [74]. The relation $\simeq$ is reflexive, symmetric and transitive so equational reasoning can be done in the usual way. We make use of heterogeneous versions of congruence, the simplest one can be stated as follows.

$$\mathsf{ap} : (f : (x : A) \to B\,x) \to a \equiv a' \to f\,a \simeq f\,a'$$

We can prove that π_2 is a congruence using this version of ap:

$$\mathsf{ap}\pi_2\,(p : \sigma \equiv \sigma') : \pi_2\,\sigma \simeq \pi_2\,\sigma' := \mathsf{ap}\,\pi_2\,p$$

Sometimes however we need a more flexible version of this congruence when not even σ and σ' have the same type. We can prove this using the eliminator for equality six times (once for each homogeneous equality and twice for each heterogeneous equality).

$$\begin{aligned}
\mathsf{ap}\pi_2' \ :\ & \Gamma \equiv \Gamma' \to \Delta \equiv \Delta' \\
& \to \{A : \mathsf{Ty}\,\Delta\}\{A' : \mathsf{Ty}\,\Delta'\} \to A \simeq A' \\
& \to \{\sigma : \mathsf{Tms}\,\Gamma\,(\Delta, A)\}\{\sigma' : \mathsf{Tms}\,\Gamma'\,(\Delta', A')\} \to \sigma \simeq \sigma' \\
& \to \pi_2\,\sigma \simeq \pi_2\,\sigma'
\end{aligned}$$

Another example is the following congruence rule for substitution extension.

$$\mathsf{ap}, : \sigma \equiv \sigma' \to t \simeq t' \to (\sigma, t) \simeq (\sigma', t')$$

In the formal development, we define such congruence rules for each constructor, as needed. In our notation for heterogeneous equational reasoning, we omit the outermost congruence rules for readability just as we did in the homogeneous case.

The advantage of heterogeneous equality is that we can forget about transports during reasoning.

$$\mathsf{untr} : \{u : P\,a\}(p : a \equiv b) \to u \simeq {}_{p*}u$$

Also, we can convert back and forth with the usual homogeneous equality ($\mathsf{from}{\simeq}$ uses the axiom K defined in section 2.2.2).

$$\begin{aligned}
&\mathsf{from}{\equiv} : (p : a \equiv b) \to u \equiv^{p} v \to u \simeq v \\
&\mathsf{from}{\simeq} : \{u\,v : A\} \to u \simeq v \to u \equiv v
\end{aligned}$$

We can use $\mathsf{from}{\equiv}$ to get a heterogeneous version of $\pi_2\beta$.

$$\pi_2\beta : \pi_2\,(\sigma, t) \simeq t := \mathsf{from}{\equiv}\,\pi_1\beta\,\pi_2\beta$$

Using this we prove the heterogeneous version of $\pi_2[]$ by heterogeneous equational reasoning.

$$\begin{aligned}
\pi_2[] \;:\;\; & (\pi_2\,\nu)[\sigma] \\
& \qquad\qquad\qquad\qquad\qquad\qquad (\mathsf{untr}\,[][]) \\
\simeq\, & {}_{[][]*}(\pi_2\,\nu)[\sigma] \\
& \qquad\qquad\qquad\qquad\qquad\qquad (\pi_2\beta^{-1}) \\
\simeq\, & \pi_2\,\big(\pi_1\,\nu \circ \sigma, {}_{[][]*}(\pi_2\,\nu)[\sigma]\big) \\
& \qquad\qquad\qquad\qquad\qquad\qquad (,\circ^{-1}) \\
\simeq\, & \pi_2\,\big((\pi_1\,\nu, \pi_2\,\nu) \circ \sigma\big) \\
& \qquad\qquad\qquad\qquad\qquad\qquad (\pi\eta) \\
\simeq\, & \pi_2\,(\nu \circ \sigma)
\end{aligned}$$

We derive the functor laws for terms following [90]. The identity law can be proved using the naturality law $,\circ$ of substitution extension. First we write the proof using extensional type theory to show the idea.

$$\begin{aligned}
[\mathsf{id}] \;:\;\; & t[\mathsf{id}] \\
& \qquad\qquad\qquad (\pi_2\beta^{-1}) \\
\equiv\, & \pi_2\,(\mathsf{id} \circ \mathsf{id}, t[\mathsf{id}]) \\
& \qquad\qquad\qquad (,\circ^{-1}) \\
\equiv\, & \pi_2\,\big((\mathsf{id}, t) \circ \mathsf{id}\big) \\
& \qquad\qquad\qquad (\circ\mathsf{id}) \\
\equiv\, & \pi_2\,(\mathsf{id}, t) \\
& \qquad\qquad\qquad (\pi_2\beta) \\
\equiv\, & t
\end{aligned}$$

The intensional proof using heterogeneous equational reasoning is more involved as the transports need to be mentioned explicitly when we use the substitution extension operator $-,-$.

$$
\begin{aligned}
[\mathsf{id}] \ : \ & t[\mathsf{id}] \\
& \qquad\qquad\qquad\qquad (\pi_2\beta^{-1}) \\
\simeq\, & \pi_2\,(\mathsf{id}, t[\mathsf{id}]) \\
& \qquad\qquad\qquad\qquad \Big(\mathsf{ap},\ (\mathsf{id}\circ^{-1})\,\Big(\mathsf{ap}\,(-[\mathsf{id}])\,\big(\mathsf{untr}\,([\mathsf{id}]^{-1})\big) \centerdot \mathsf{untr}\,[][]\Big)\Big) \\
\simeq\, & \pi_2\,\Big(\mathsf{id}\circ\mathsf{id}, {}_{[][]*}\big(({}_{[\mathsf{id}]^{-1}*}t)[\mathsf{id}]\big)\Big) \\
& \qquad\qquad\qquad\qquad (,\circ^{-1}) \\
\simeq\, & \pi_2\,\big((\mathsf{id}, {}_{[\mathsf{id}]^{-1}*}t)\circ\mathsf{id}\big) \\
& \qquad\qquad\qquad\qquad (\circ\mathsf{id}) \\
\simeq\, & \pi_2\,(\mathsf{id}, {}_{[\mathsf{id}]^{-1}*}t) \\
& \qquad\qquad\qquad\qquad (\pi_2\beta) \\
\simeq\, & {}_{[\mathsf{id}]^{-1}*}t \\
& \qquad\qquad\qquad\qquad \big(\mathsf{untr}\,([\mathsf{id}]^{-1})^{-1}\big) \\
\simeq\, & t
\end{aligned}
$$

We only write down the proof for composition using extensional type theory.

$$
\begin{aligned}
[][] \ : \ & t[\sigma][\nu] \\
& \qquad\qquad\qquad\qquad (\pi_2\beta^{-1}) \\
\equiv\, & \pi_2\,\big((\mathsf{id}\circ\sigma)\circ\nu, t[\sigma][\nu]\big) \\
& \qquad\qquad\qquad\qquad (,\circ^{-1}) \\
\equiv\, & \pi_2\,\big((\mathsf{id}\circ\sigma, t[\sigma])\circ\nu\big) \\
& \qquad\qquad\qquad\qquad (,\circ^{-1}) \\
\equiv\, & \pi_2\,\Big(\big((\mathsf{id}, t)\circ\sigma\big)\circ\nu\Big) \\
& \qquad\qquad\qquad\qquad (\circ\circ) \\
\equiv\, & \pi_2\,\big((\mathsf{id}, t)\circ(\sigma\circ\nu)\big) \\
& \qquad\qquad\qquad\qquad (,\circ) \\
\equiv\, & \pi_2\,\big(\mathsf{id}\circ(\sigma\circ\nu), t[\sigma\circ\nu]\big) \\
& \qquad\qquad\qquad\qquad (\pi_2\beta) \\
\equiv\, & t[\sigma\circ\nu]
\end{aligned}
$$

With all this machinery at hand we can reproduce the example derivation trees in figure 3.2.

Example A is the polymorphic identity function. We view the base type U as a universe for this example. The first argument of the function is a code for a type and we need to use El to turn it into a type. Here we also need to use transports at multiple places.

$$\mathsf{lam}\left(\mathsf{lam}\left({}_{\mathsf{El}[]*}\mathsf{vz}\right)\right) : \mathsf{Tm}\ \cdot\ \left(\Pi\,\mathsf{U}\left(\Pi\left(\mathsf{El}\left({}_{\mathsf{U}[]*}\mathsf{vz}\right)\right)\left(\mathsf{El}\left({}_{\mathsf{U}[]*}({}_{\mathsf{U}[]*}\mathsf{vz})[\mathsf{wk}]\right)\right)\right)\right)$$

Example B is predicate space where the indexing type of the predicate is a variable in the context. We derive a type, not a term of type U because our universe U is empty.

$$\Pi\left(\mathsf{El}\left({}_{\mathsf{U}[]*}\mathsf{vz}\right)\right)\mathsf{U} : \mathsf{Ty}\,(\cdot, \mathsf{U})$$

In this example Agda's mechanism for implicit arguments figures out that the codomain of the function will be in the context $\cdot, \mathsf{U}, \mathsf{El}\,({}_{\mathsf{U}[]*}\mathsf{vz})$ so we don't need to write down this context derivation by hand.

To implement example C, we first define the non dependent function space with its usual application and substitution law.

$$\begin{aligned}
&(A : \mathsf{Ty}\,\Gamma) \Rightarrow (B : \mathsf{Ty}\,\Gamma) : \mathsf{Ty}\,\Gamma := \Pi\,A\,(B[\mathsf{wk}])\\
&(t : \mathsf{Tm}\,\Gamma\,(A \Rightarrow B))\$(u : \mathsf{Tm}\,\Gamma\,A) : \mathsf{Tm}\,\Gamma\,B\\
&\qquad\quad := {}_{\left([][]\blacksquare\mathsf{ap}\,A[-]\,(\pi_1\circ\blacksquare\mathsf{ap}\,\pi_1\,\mathsf{id}\circ\blacksquare\pi_1\beta)\blacksquare[\mathsf{id}]\right)*}(\mathsf{app}\,t)[\langle u\rangle]\\
&\Rightarrow[] : (A \Rightarrow B)[\sigma] \equiv A[\sigma] \Rightarrow B[\sigma]\\
&\qquad\quad := \Pi[] \blacksquare \mathsf{ap}\left(\Pi\,(A[\sigma])\right)\left([][] \blacksquare \mathsf{ap}\,(B[-])\,(\pi_1 \circ \blacksquare \mathsf{ap}\,\pi_1\,\mathsf{id} \circ \blacksquare \pi_1\beta) \blacksquare [][]^{-1}\right)
\end{aligned}$$

First define the projection p in figure 3.2 and denote it by N. This is just the type computed using El from the second De Bruijn index.

$$N : \mathsf{Ty}\left(\cdot, \mathsf{U}, \mathsf{El}\left({}_{\mathsf{U}[]*}\mathsf{vz}\right)\right) := \mathsf{El}\left({}_{([][]\blacksquare\mathsf{U}[])*}\mathsf{vs}\,\mathsf{vz}\right)$$

The context Γ is defined as follows.

$$\Gamma : \mathsf{Con} := \cdot, \mathsf{U}, \mathsf{El}\left({}_{\mathsf{U}[]*}\mathsf{vz}\right), N \Rightarrow N$$

The projection z is defined as the second De Bruijn index in context Γ and we need to use transport.

$$\mathsf{z} : \mathsf{Tm}\,\Gamma\,(N[\mathsf{wk}]) := {}_{([][]\centerdot\mathsf{El}[]\centerdot\mathsf{ap}\,\mathsf{El}\,(\mathsf{from}{\simeq}\,\mathsf{r})\centerdot\mathsf{El}[]^{-1})*}\mathsf{vs}\,\mathsf{vz}$$

The proof of r is given as follows.

$$\begin{array}{ll}
\mathsf{r}\ :\ \ {}_{\mathsf{U}[]*}({}_{\mathsf{U}[]*}\mathsf{vz})[\mathsf{wk}\circ\mathsf{wk}] & \\
 & (\mathsf{untr}\,\mathsf{U}[]^{-1}) \\
\quad\simeq ({}_{\mathsf{U}[]*}\mathsf{vz})[\mathsf{wk}\circ\mathsf{wk}] & \\
 & \big(\mathsf{ap}\,(-[\mathsf{wk}\circ\mathsf{wk}])\,(\mathsf{untr}\,\mathsf{U}[]^{-1})\big) \\
\quad\simeq \mathsf{vz}[\mathsf{wk}\circ\mathsf{wk}] & \\
 & ([][]^{-1}) \\
\quad\simeq (\mathsf{vs}\,\mathsf{vz})[\mathsf{wk}] & \\
 & \Big(\mathsf{ap}\,(-[\mathsf{wk}])\,\big(\mathsf{untr}\,([][]\centerdot\mathsf{U}[])\big)\Big) \\
\quad\simeq ({}_{([][]\centerdot\mathsf{U}[])*}\mathsf{vs}\,\mathsf{vz})[\mathsf{wk}] & \\
 & (\mathsf{untr}\,\mathsf{U}[]) \\
\quad\simeq {}_{\mathsf{U}[]*}({}_{([][]\centerdot\mathsf{U}[])*}\mathsf{vs}\,\mathsf{vz})[\mathsf{wk}] &
\end{array}$$

By proving the equality

$$\mathsf{s} : N[\mathsf{wk}] \equiv \mathsf{El}\,\big({}_{([][]\centerdot[][]\centerdot\mathsf{U}[])*}\mathsf{vs}\,(\mathsf{vs}\,\mathsf{vz})\big),$$

we can define sz and finally ssz which was our final goal.

$$\begin{aligned}
\mathsf{sz}\ &: \mathsf{Tm}\,\Gamma\,\Big(\mathsf{El}\,\big({}_{([][]\centerdot[][]\centerdot\mathsf{U}[])*}\mathsf{vs}\,(\mathsf{vs}\,\mathsf{vz})\big)\Big) := \big({}_{(\Rightarrow[]\centerdot\mathsf{ap}\,(\lambda z.N[\mathsf{wk}]\Rightarrow z)\,\mathsf{s})*}\mathsf{vz}\big)\,\$\,\mathsf{z} \\
\mathsf{ssz} &: \mathsf{Tm}\,\Gamma\,\Big(\mathsf{El}\,\big({}_{([][]\centerdot[][]\centerdot\mathsf{U}[])*}\mathsf{vs}\,(\mathsf{vs}\,\mathsf{vz})\big)\Big) := \big({}_{(\Rightarrow[]\centerdot\mathsf{ap}\,(\lambda z.z\Rightarrow z)\,\mathsf{s})*}\mathsf{vz}\big)\,\$\,\mathsf{sz}
\end{aligned}$$

3.4 The recursor

The recursor is determined by the constructors of the syntax. We collect the arguments of the recursor into a record called Model given below. We call an element of this record a model of type theory, that is, a sound semantics. Every syntactic construct has a semantic counterpart in this record including the equations, the latter ensure soundness.

First we list the fields for the substitution calculus. These have the same type as the type formation rules and constructors but with $^{\mathsf{M}}$ indices to distinguish from the constructor names.

$$\begin{array}{ll}
\mathsf{record\ Model} & : \mathsf{Set}_1 \\
\quad \mathsf{Con}^{\mathsf{M}} & : \mathsf{Set} \\
\quad \mathsf{Ty}^{\mathsf{M}} & : \mathsf{Con}^{\mathsf{M}} \to \mathsf{Set} \\
\quad \mathsf{Tms}^{\mathsf{M}} & : \mathsf{Con}^{\mathsf{M}} \to \mathsf{Con}^{\mathsf{M}} \to \mathsf{Set} \\
\quad \mathsf{Tm}^{\mathsf{M}} & : (\Gamma^M : \mathsf{Con}^{\mathsf{M}}) \to \mathsf{Ty}^{\mathsf{M}}\,\Gamma^M \to \mathsf{Set} \\
\quad \cdot^{\mathsf{M}} & : \mathsf{Con}^{\mathsf{M}} \\
\quad -,^{\mathsf{M}} - & : (\Gamma^M : \mathsf{Con}^{\mathsf{M}}) \to \mathsf{Ty}^{\mathsf{M}}\,\Gamma^M \to \mathsf{Con}^{\mathsf{M}} \\
\quad -[-]^{\mathsf{M}} & : \mathsf{Ty}^{\mathsf{M}}\,\Theta^M \to \mathsf{Tms}^{\mathsf{M}}\,\Gamma^M\,\Theta^M \to \mathsf{Ty}^{\mathsf{M}}\,\Gamma^M \\
\quad - \circ^{\mathsf{M}} - & : \mathsf{Tms}^{\mathsf{M}}\,\Theta^M\,\Delta^M \to \mathsf{Tms}^{\mathsf{M}}\,\Gamma^M\,\Theta^M \to \mathsf{Tms}^{\mathsf{M}}\,\Gamma^M\,\Delta^M \\
\quad \mathsf{id}^{\mathsf{M}} & : \mathsf{Tms}^{\mathsf{M}}\,\Gamma^M\,\Gamma^M \\
\quad \epsilon^{\mathsf{M}} & : \mathsf{Tms}^{\mathsf{M}}\,\Gamma^M\,\cdot^{\mathsf{M}} \\
\quad -,^{\mathsf{M}} - & : (\sigma^M : \mathsf{Tms}^{\mathsf{M}}\,\Gamma^M\,\Delta^M) \to \mathsf{Tm}^{\mathsf{M}}\,\Gamma^M\,A^M[\sigma^M]^{\mathsf{M}} \to \mathsf{Tms}^{\mathsf{M}}\,\Gamma^M\,(\Delta^M,^{\mathsf{M}} A^M) \\
\quad {\pi_1}^{\mathsf{M}} & : \mathsf{Tms}^{\mathsf{M}}\,\Gamma^M\,(\Delta^M,^{\mathsf{M}} A^M) \to \mathsf{Tms}^{\mathsf{M}}\,\Gamma^M\,\Delta^M \\
\quad -[-]^{\mathsf{M}} & : \mathsf{Tm}^{\mathsf{M}}\,\Theta^M\,A^M \to (\sigma^M : \mathsf{Tms}^{\mathsf{M}}\,\Gamma^M\,\Theta^M) \to \mathsf{Tm}^{\mathsf{M}}\,\Gamma^M\,A^M[\sigma^M]^{\mathsf{M}} \\
\quad {\pi_2}^{\mathsf{M}} & : (\sigma^M : \mathsf{Tms}^{\mathsf{M}}\,\Gamma^M\,(\Delta^M,^{\mathsf{M}} A^M)) \to \mathsf{Tm}^{\mathsf{M}}\,\Gamma^M\,A^M[{\pi_1}^{\mathsf{M}}\,\sigma^M]^{\mathsf{M}} \\
\quad [][]^{\mathsf{M}} & : A^M[\sigma^M]^{\mathsf{M}}[\nu^M]^{\mathsf{M}} \equiv A^M[\sigma^M \circ^{\mathsf{M}} \nu^M]^{\mathsf{M}} \\
\quad [\mathsf{id}]^{\mathsf{M}} & : A^M[\mathsf{id}^{\mathsf{M}}]^{\mathsf{M}} \equiv A^M \\
\quad \circ\circ^{\mathsf{M}} & : (\sigma^M \circ^{\mathsf{M}} \nu^M) \circ^{\mathsf{M}} \delta^M \equiv \sigma^M \circ^{\mathsf{M}} (\nu^M \circ^{\mathsf{M}} \delta^M) \\
\quad \mathsf{id}\circ^{\mathsf{M}} & : \mathsf{id}^{\mathsf{M}} \circ^{\mathsf{M}} \sigma^M \equiv \sigma^M \\
\quad \circ\mathsf{id}^{\mathsf{M}} & : \sigma^M \circ^{\mathsf{M}} \mathsf{id}^{\mathsf{M}} \equiv \sigma^M \\
\quad \epsilon\eta^{\mathsf{M}} & : \{\sigma^M : \mathsf{Tms}^{\mathsf{M}}\,\Gamma^M\,\cdot^{\mathsf{M}}\} \to \sigma^M \equiv \epsilon^{\mathsf{M}} \\
\quad \pi_1\beta^{\mathsf{M}} & : {\pi_1}^{\mathsf{M}}\,(\sigma^M,^{\mathsf{M}} t^M) \equiv \sigma^M \\
\quad \pi\eta^{\mathsf{M}} & : ({\pi_1}^{\mathsf{M}}\,\sigma^M,^{\mathsf{M}} {\pi_2}^{\mathsf{M}}\,\sigma^M) \equiv \sigma^M \\
\quad ,\circ^{\mathsf{M}} & : (\nu^M,^{\mathsf{M}} t^M) \circ^{\mathsf{M}} \sigma^M \equiv (\nu^M \circ^{\mathsf{M}} \sigma^M),^{\mathsf{M}} ({}_{[][]^{\mathsf{M}}*} t^M[\sigma^M]^{\mathsf{M}}) \\
\quad \pi_2\beta^{\mathsf{M}} & : {\pi_2}^{\mathsf{M}}\,(\sigma^M,^{\mathsf{M}} t^M) \equiv^{\pi_1\beta^{\mathsf{M}}} t^M
\end{array}$$

The fields for the base type and base family are given below.

$$
\begin{array}{ll}
\multicolumn{2}{l}{\mathsf{record\ Model{+}}} \\
\quad \mathsf{U}^{\mathsf{M}} & : \mathsf{Ty}^{\mathsf{M}}\,\Gamma^{M} \\
\quad \mathsf{El}^{\mathsf{M}} & : \mathsf{Tm}^{\mathsf{M}}\,\Gamma^{M}\,\mathsf{U}^{\mathsf{M}} \to \mathsf{Ty}^{\mathsf{M}}\,\Gamma^{M} \\
\quad \mathsf{U}[]^{\mathsf{M}} & : \mathsf{U}^{\mathsf{M}}[\sigma^{M}]^{\mathsf{M}} \equiv \mathsf{U}^{\mathsf{M}} \\
\quad \mathsf{El}[]^{\mathsf{M}} & : (\mathsf{El}^{\mathsf{M}}\,\hat{A}^{M})[\sigma^{M}]^{\mathsf{M}} \equiv \mathsf{El}^{\mathsf{M}}\,({}_{\mathsf{U}[]^{\mathsf{M}}*}\hat{A}^{M}[\sigma^{M}]^{\mathsf{M}})
\end{array}
$$

To define the fields for the function space first we need the semantic counterpart of the lifting function.

$$
\begin{array}{l}
(\sigma^{M} : \mathsf{Tms}^{\mathsf{M}}\,\Gamma^{M}\,\Delta^{M}) \uparrow^{\mathsf{M}} A^{M} : \mathsf{Tms}^{\mathsf{M}}\,(\Gamma^{M},^{\mathsf{M}} A^{M}[\sigma^{M}]^{\mathsf{M}})\,(\Delta^{M},^{\mathsf{M}} A^{M}) \\
\qquad := (\sigma^{M} \circ^{\mathsf{M}} \pi_1^{\mathsf{M}}\,\mathsf{id}^{\mathsf{M}}),^{\mathsf{M}} ({}_{[][]^{\mathsf{M}}*}\pi_2^{\mathsf{M}}\,\mathsf{id}^{\mathsf{M}})
\end{array}
$$

Now the fields for the function space are given as follows.

$$
\begin{array}{ll}
\multicolumn{2}{l}{\mathsf{record\ Model{+}}} \\
\quad \Pi^{\mathsf{M}} & : (A^{M} : \mathsf{Ty}^{\mathsf{M}}\,\Gamma^{M}) \to \mathsf{Ty}^{\mathsf{M}}\,(\Gamma^{M},^{\mathsf{M}} A^{M}) \to \mathsf{Ty}^{\mathsf{M}}\,\Gamma^{M} \\
\quad \Pi[]^{\mathsf{M}} & : (\Pi^{\mathsf{M}}\,A^{M}\,B^{M})[\sigma^{M}]^{\mathsf{M}} \equiv \Pi^{\mathsf{M}}\,(A^{M}[\sigma^{M}]^{\mathsf{M}})\,(B^{M}[\sigma^{M} \uparrow^{\mathsf{M}} A^{M}]^{\mathsf{M}}) \\
\quad \mathsf{lam}^{\mathsf{M}} & : \mathsf{Tm}^{\mathsf{M}}\,(\Gamma^{M},^{\mathsf{M}} A^{M})\,B^{M} \to \mathsf{Tm}^{\mathsf{M}}\,\Gamma^{M}\,(\Pi^{\mathsf{M}}\,A^{M}\,B^{M}) \\
\quad \mathsf{app}^{\mathsf{M}} & : \mathsf{Tm}^{\mathsf{M}}\,\Gamma^{M}\,(\Pi^{\mathsf{M}}\,A^{M}\,B^{M}) \to \mathsf{Tm}^{\mathsf{M}}\,(\Gamma^{M},^{\mathsf{M}} A^{M})\,B^{M} \\
\quad \Pi\beta^{\mathsf{M}} & : \mathsf{app}^{\mathsf{M}}\,(\mathsf{lam}^{\mathsf{M}}\,t^{M}) \equiv t^{M} \\
\quad \Pi\eta^{\mathsf{M}} & : \mathsf{lam}^{\mathsf{M}}\,(\mathsf{app}^{\mathsf{M}}\,t^{M}) \equiv t^{M} \\
\quad \mathsf{lam}[]^{\mathsf{M}} & : (\mathsf{lam}^{\mathsf{M}}\,t^{M})[\sigma^{M}]^{\mathsf{M}} \equiv^{\Pi[]^{\mathsf{M}}} \mathsf{lam}^{\mathsf{M}}\,(t^{M}[\sigma^{M} \uparrow^{\mathsf{M}} A^{M}])
\end{array}
$$

The recursor is parameterised by a model and it satisfies the equations given below.

$$
\begin{array}{llll}
\multicolumn{4}{l}{\mathsf{module\ Rec}\,(m : \mathsf{Model})} \\
\multicolumn{4}{l}{\quad \mathsf{open\ Model}\,m} \\
\quad \mathsf{RecCon} & : \mathsf{Con} & \to & \mathsf{Con}^{\mathsf{M}} \\
\quad \mathsf{RecTy} & : \mathsf{Ty}\,\Gamma & \to & \mathsf{Ty}^{\mathsf{M}}\,(\mathsf{RecCon}\,\Gamma) \\
\quad \mathsf{RecTms} & : \mathsf{Tms}\,\Gamma\,\Delta & \to & \mathsf{Tms}^{\mathsf{M}}\,(\mathsf{RecCon}\,\Gamma)\,(\mathsf{RecCon}\,\Delta) \\
\quad \mathsf{RecTm} & : \mathsf{Tm}\,\Gamma\,A & \to & \mathsf{Tm}^{\mathsf{M}}\,(\mathsf{RecCon}\,\Gamma)\,(\mathsf{RecTy}\,A)
\end{array}
$$

$$\begin{array}{lll}
\mathsf{RecCon} & \cdot & = \cdot^{\mathsf{M}} \\
\mathsf{RecCon} & (\Gamma, A) & = (\mathsf{RecCon}\,\Gamma ,^{\mathsf{M}} \mathsf{RecTy}\,A) \\
\mathsf{RecTy} & (A[\sigma]) & = \mathsf{RecTy}\,A[\mathsf{RecTms}\,\sigma]^{\mathsf{M}} \\
\mathsf{RecTms} & (\sigma \circ \nu) & = \mathsf{RecTms}\,\sigma \circ^{\mathsf{M}} \mathsf{RecTms}\,\nu \\
\mathsf{RecTms} & \mathsf{id} & = \mathsf{id}^{\mathsf{M}} \\
\mathsf{RecTms} & \epsilon & = \epsilon^{\mathsf{M}} \\
\mathsf{RecTms} & (\sigma, t) & = (\mathsf{RecTms}\,\sigma ,^{\mathsf{M}} \mathsf{RecTm}\,t) \\
\mathsf{RecTms} & (\pi_1\,\sigma) & = \pi_1^{\mathsf{M}}\,(\mathsf{RecTms}\,\sigma) \\
\mathsf{RecTm} & (t[\sigma]) & = (\mathsf{RecTm}\,t)[\mathsf{RecTms}\,\sigma]^{\mathsf{M}} \\
\mathsf{RecTm} & (\pi_2\,\sigma) & = \pi_2^{\mathsf{M}}\,(\mathsf{RecTms}\,\sigma) \\
\mathsf{RecTy} & \mathsf{U} & = \mathsf{U}^{\mathsf{M}} \\
\mathsf{RecTy} & (\mathsf{El}\,\hat{A}) & = \mathsf{El}^{\mathsf{M}}\,(\mathsf{RecTm}\,\hat{A}) \\
\mathsf{RecTy} & (\Pi\,A\,B) & = \Pi^{\mathsf{M}}\,(\mathsf{RecTy}\,A)\,(\mathsf{RecTy}\,B) \\
\mathsf{RecTm} & (\mathsf{lam}\,t) & = \mathsf{lam}^{\mathsf{M}}\,(\mathsf{RecTm}\,t) \\
\mathsf{RecTm} & (\mathsf{app}\,t) & = \mathsf{app}^{\mathsf{M}}\,(\mathsf{RecTm}\,t)
\end{array}$$

It is possible to define the category of models by taking objects to be elements of Model while a morphism between two models is given by 4 mutually defined functions which have the same types as the 4 Rec... functions of the recursor (but go from one model to another, not from the syntax to another one) which satisfy the same equations that the recursor satisfies but it is enough to satisfy them propositionally. With this view in mind the recursor states that the syntax (the term model, see below) is the (weakly) initial model. In section 4.3 we give a method for deriving these equations from the syntax.

Our definition of model of type theory is equivalent to categories with families by Dybjer (CwF), a standard notion of model of type theory. His requirements in Definition 1 of [48] are met by the following observations. Here we omit the $^{\mathsf{M}}$ indices for readability, but we are talking about an arbitrary model, not the syntax.

- We have a category $\mathcal{C}$. The objects are given by Con, the morphisms by Tms, and we have the categorical operations $- \circ -$, id and laws ($\circ\circ$, $\mathsf{id}\circ$, $\circ\mathsf{id}$).
- We have a functor $T : \mathcal{C}^{\mathsf{op}} \to \mathsf{Fam}$. The object part is given by Ty and Tm, the morphism part by the $-[-]$ operations and we have the functor

laws $[][]$, $[\mathsf{id}]$. The functor laws for the Tm part are derivable, see previous section.

- $\mathcal{C}$ has a terminal object $\cdot$ and terminality is given by ϵ, $\epsilon\eta$.

- We express the comprehension operation as a natural isomorphism.

  Dybjer's notion of comprehension requires $-,-$ operations for context and substitution extensions which have the same types as ours together with $\mathsf{p} : \mathsf{Tms}\,(\Gamma, A)\,\Gamma$ and $\mathsf{q} : \mathsf{Tm}\,(\Gamma, A)\,A[\mathsf{p}]$ operators such that $\mathsf{p} \circ (\sigma, t) \equiv \sigma$ and $\mathsf{q}[\sigma, t] \equiv t$. In addition, he has a uniqueness condition which states that given a ν s.t. $\mathsf{p} \circ \nu \equiv \sigma$ and $\mathsf{q}[\nu] \equiv t$, we have that $\nu \equiv (\sigma, t)$.

  We can show that the two notions are isomorphic by defining two maps and showing that their compositions are identities.

  Both maps are identity on the $-,-$ operations. Starting from our notion, we can define $\mathsf{p} := \pi_1\,\mathsf{id}$ and $\mathsf{q} := \pi_2\,\mathsf{id}$ and we need to check the uniqueness condition. Given ν s.t. $\pi_1\,\mathsf{id} \circ \nu \equiv \sigma$ and $(\pi_2\mathsf{id})[\nu] \equiv t$ we have $\nu \equiv (\pi_1\,\nu, \pi_2\,\nu) \equiv (\pi_1\,\mathsf{id} \circ \nu, (\pi_2\,\mathsf{id})[\nu]) \equiv (\sigma, t)$. Starting from Dybjer's notion, we define $\pi_1\,\sigma := \mathsf{p} \circ \sigma$ and $\pi_2\,\sigma := \mathsf{q}[\sigma]$. We check the equations:

  $$\begin{aligned}
  &\pi_1\beta : \pi_1\,(\sigma, t) = \mathsf{p} \circ (\sigma, t) \equiv \sigma \\
  &\pi_2\beta : \pi_2\,(\sigma, t) = \mathsf{q}[\sigma, t] \equiv t \\
  &\pi\eta\ \ : (\pi_1\,\sigma, \pi_2\,\sigma) = (\mathsf{p} \circ \sigma, \mathsf{q}[\sigma]) \equiv \sigma \\
  &\qquad \text{by uniqueness as we have } \mathsf{p} \circ \sigma \equiv \mathsf{p} \circ \sigma \text{ and } \mathsf{q}[\sigma] \equiv \mathsf{q}[\sigma] \\
  &,\circ\ \ \ : (\nu, t) \circ \sigma \equiv (\nu \circ \sigma, t[\sigma]) \\
  &\qquad \text{by uniqueness as we have } \mathsf{p} \circ \big((\nu, t) \circ \sigma\big) \equiv \big(\mathsf{p} \circ (\nu, t)\big) \circ \sigma \equiv \nu \circ \sigma \\
  &\qquad \text{and } \mathsf{q}[(\nu, t) \circ \sigma] \equiv \mathsf{q}[(\nu, t)][\sigma] \equiv t[\sigma]
  \end{aligned}$$

  For checking that compositions are identities it is enough to check the projections because the equations will be equal by K. We have $\pi_1\,\sigma \mapsto \pi_1\,\mathsf{id}{\circ}\sigma \equiv \pi_1\,\sigma$ by $\pi_1\circ$ and $\mathsf{id}\circ$ and $\pi_2\,\sigma \mapsto (\pi_2\,\mathsf{id})[\sigma] \equiv \pi_2\,\sigma$ by $\pi_2[]$ and $\mathsf{id}\circ$. In the other direction we have $\mathsf{p} \mapsto \mathsf{p} \circ \mathsf{id} \equiv \mathsf{p}$ and $\mathsf{q} \mapsto \mathsf{q}[\mathsf{id}] \equiv \mathsf{q}$ by the identity laws.

The simplest model that we can define is the term model which interprets every syntactic construct by itself. We only list the first few fields of the record

Model.

$$
\begin{aligned}
\mathsf{Con}^{\mathsf{M}} &:= \mathsf{Con} \\
\mathsf{Ty}^{\mathsf{M}} &:= \mathsf{Ty} \\
\mathsf{Tms}^{\mathsf{M}} &:= \mathsf{Tms} \\
\mathsf{Tm}^{\mathsf{M}} &:= \mathsf{Tm} \\
\cdot^{\mathsf{M}} &:= \cdot \\
-,^{\mathsf{M}} - &:= -, - \\
-[-]^{\mathsf{M}} &:= -[-] \\
\ldots
\end{aligned}
$$

3.4.1 Disjointness of context and type constructors

As a simple example of using the recursor, we define another special recursor which only works for contexts. RecCon' can be used for defining a function from Con which only depends on the length of the context but not the types in the context.

$$
\begin{aligned}
&\mathsf{RecCon}'(Con^M : \mathsf{Set})(\cdot^M : Con^M)(-,^M - : Con^M \to Con^M) : \mathsf{Con} \to Con^M \\
&\quad := \mathsf{Rec.RecCon}\ (\ \mathsf{Con}^{\mathsf{M}} := Con^M \\
&\qquad\qquad , \mathsf{Ty}^{\mathsf{M}} := \lambda\_.\top \\
&\qquad\qquad , \mathsf{Tms}^{\mathsf{M}} := \lambda\_\_.\top \\
&\qquad\qquad , \mathsf{Tm}^{\mathsf{M}} := \lambda\_\_.\top \\
&\qquad\qquad , \cdot^{\mathsf{M}} := \cdot^M \\
&\qquad\qquad , -,^{\mathsf{M}} - := -,^M - \\
&\qquad\qquad , -[-]^{\mathsf{M}} := \lambda\_\_.\mathsf{tt} \\
&\qquad\qquad , \ldots \\
&\qquad\qquad , [\mathsf{id}] := \mathsf{refl} \\
&\qquad\qquad , \ldots)
\end{aligned}
$$

For this we need an interpretation $\cdot^M$ of the empty context and the interpretation $-,^M -$ of context extension (which does not depend on the type that the context was extended with). To define RecCon' we specify a model where contexts are mapped to Con^M and types, substitutions and terms are just mapped to the one-element type $\top$. Hence, the methods for these will be just the element tt of this type and all the equalities will be reflexivities. Then we use RecCon on this model to implement RecCon'. E.g. the length of a context Γ is calculated by $\mathsf{RecCon}'\,\mathbb{N}\,\mathsf{zero}\,\mathsf{suc}\,\Gamma$.

Another usage of RecCon' is proving the disjointness of $\cdot$ and Γ, A for any Γ and A. We eliminate into Set (we use large elimination) and we map $\cdot$ to $\top$ and an extended context to $\bot$. Now given a proof that $\cdot \equiv \Gamma, A$ we can transport an element of type $\top$ (the interpretation of $\cdot$) along this proof to $\bot$ (the interpretation of Γ, A).

$$\mathsf{disj}_{\cdot\,(\Gamma,A)}\,(p : \cdot \equiv \Gamma, A) : \bot := \mathsf{coe}\,\Big(\mathsf{ap}\,\big(\mathsf{RecCon}'\,\mathsf{Set}\,\bot\,(\lambda\,x.\top)\big)\,p\Big)\,\mathsf{tt}$$

We use another model to show the disjointness of the type constructors Π and U. The idea is that we interpret types as elements of Set, Π is interpreted as $\top$ and U is interpreted as $\bot$. Now, given an equality between $\Pi\,A\,B$ and U, we can transport an element of $\mathsf{RecTy}\,(\Pi\,A\,B) = \top$ to $\mathsf{RecTy}\,\mathsf{U} = \bot$. The only complication is that the semantic versions of the equalities $\Pi[]$ and $\mathsf{U}[]$ need to be satisfied: this can be achieved by taking the interpretation of $-[-]$ for types to be the identity function which simply ignores the substitution.

The motives of this model and the methods for types are given as follows.

$$\begin{array}{llll}
\mathsf{Con}^{\mathsf{M}} & := \top & X[\_]^{\mathsf{M}} & := X \\
\mathsf{Ty}^{\mathsf{M}}\,\_ & := \mathsf{Set} & \mathsf{U}^{\mathsf{M}} & := \bot \\
\mathsf{Tms}^{\mathsf{M}}\,\_\,\_ & := \top & \mathsf{El}^{\mathsf{M}}\,\_ & := \top \\
\mathsf{Tm}^{\mathsf{M}}\,\_\,\_ & := \top & \Pi^{\mathsf{M}}\,\_\,\_ & := \top
\end{array}$$

The methods for contexts, substitutions and terms just return tt. All the equality methods are refl. Denoting this model by M, disjointness of Π and U is given by the following function.

$$\mathsf{disj}_{(\Pi\,A\,B)\,\mathsf{U}}\,(p : \Pi\,A\,B \equiv \mathsf{U}) : \bot := \mathsf{coe}\,\big(\mathsf{ap}\,(\mathsf{RecTy}\,M)\,p\big)\,\mathsf{tt}$$

3.5 The elimination principle

The elimination principle allows the definition of a "dependent model" where the semantic constructs can depend on the elements of the syntax. For example, contexts are not just modelled by a type, but a type depending on a particular context. An example of such a model is the logical predicate interpretation in the next chapter. The recursor can be defined as a special case of the eliminator.

We list the motives and methods for the eliminator for the core substitution calculus. We derived these fields from the constructors using the method in section 4.3.

$$
\begin{array}{ll}
\multicolumn{2}{l}{\mathsf{record\ DModel} : \mathsf{Set}_1} \\
\quad \mathsf{Con}^{\mathsf{M}} & : \mathsf{Con} \to \mathsf{Set} \\
\quad \mathsf{Ty}^{\mathsf{M}} & : \mathsf{Con}^{\mathsf{M}}\,\Gamma \to \mathsf{Ty}\,\Gamma \to \mathsf{Set} \\
\quad \mathsf{Tms}^{\mathsf{M}} & : \mathsf{Con}^{\mathsf{M}}\,\Gamma \to \mathsf{Con}^{\mathsf{M}}\,\Delta \to \mathsf{Tms}\,\Gamma\,\Delta \to \mathsf{Set} \\
\quad \mathsf{Tm}^{\mathsf{M}} & : (\Gamma^M : \mathsf{Con}^{\mathsf{M}}\,\Gamma) \to \mathsf{Ty}^{\mathsf{M}}\,\Gamma^M\,A \to \mathsf{Tm}\,\Gamma\,A \to \mathsf{Set} \\
\quad .^{\mathsf{M}} & : \mathsf{Con}^{\mathsf{M}}\,\cdot \\
\quad -,^{\mathsf{M}}- & : (\Gamma^M : \mathsf{Con}^{\mathsf{M}}\,\Gamma) \to \mathsf{Ty}^{\mathsf{M}}\,\Gamma^M\,A \to \mathsf{Con}^{\mathsf{M}}\,(\Gamma, A) \\
\quad -[-]^{\mathsf{M}} & : \mathsf{Ty}^{\mathsf{M}}\,\Theta^M\,A \to \mathsf{Tms}^{\mathsf{M}}\,\Gamma^M\,\Theta^M\,\sigma \to \mathsf{Ty}^{\mathsf{M}}\,\Gamma^M\,(A[\sigma]) \\
\quad - \circ^{\mathsf{M}} - & : \mathsf{Tms}^{\mathsf{M}}\,\Theta^M\,\Delta^M\,\sigma \to \mathsf{Tms}^{\mathsf{M}}\,\Gamma^M\,\Theta^M\,\nu \to \mathsf{Tms}^{\mathsf{M}}\,\Gamma^M\,\Delta^M\,(\sigma \circ \nu) \\
\quad \mathsf{id}^{\mathsf{M}} & : \mathsf{Tms}^{\mathsf{M}}\,\Gamma^M\,\Gamma^M\,\mathsf{id} \\
\quad \epsilon^{\mathsf{M}} & : \mathsf{Tms}^{\mathsf{M}}\,\Gamma^M\ \cdot^{\mathsf{M}}\ \epsilon \\
\quad -,^{\mathsf{M}}- & : (\sigma^M : \mathsf{Tms}^{\mathsf{M}}\,\Gamma^M\,\Delta^M\,\sigma) \to \mathsf{Tm}^{\mathsf{M}}\,\Gamma^M\,A^M[\sigma^M]^{\mathsf{M}}\,t \\
 & \to \mathsf{Tms}^{\mathsf{M}}\,\Gamma^M\,(\Delta^M,^{\mathsf{M}} A^M)\,(\sigma, t) \\
\quad {\pi_1}^{\mathsf{M}} & : \mathsf{Tms}^{\mathsf{M}}\,\Gamma^M\,(\Delta^M,^{\mathsf{M}} A^M)\,\sigma \to \mathsf{Tms}^{\mathsf{M}}\,\Gamma^M\,\Delta^M\,(\pi_1\,\sigma) \\
\quad -[-]^{\mathsf{M}} & : \mathsf{Tm}^{\mathsf{M}}\,\Theta^M\,A^M\,t \to (\sigma^M : \mathsf{Tms}^{\mathsf{M}}\,\Gamma^M\,\Theta^M\,\sigma) \\
 & \to \mathsf{Tm}^{\mathsf{M}}\,\Gamma^M\,A^M[\sigma^M]^{\mathsf{M}}\,(t[\sigma]) \\
\quad {\pi_2}^{\mathsf{M}} & : (\sigma^M : \mathsf{Tms}^{\mathsf{M}}\,\Gamma^M\,(\Delta^M,^{\mathsf{M}} A^M)\,\sigma) \\
 & \to \mathsf{Tm}^{\mathsf{M}}\,\Gamma^M\,A^M[{\pi_1}^{\mathsf{M}}\,\sigma^M]^{\mathsf{M}}\,(\pi_2\,\sigma) \\
\quad [][]^{\mathsf{M}} & : A^M[\sigma^M]^{\mathsf{M}}[\nu^M]^{\mathsf{M}} \equiv^{[][]} A^M[\sigma^M \circ^{\mathsf{M}} \nu^M]^{\mathsf{M}} \\
\quad [\mathsf{id}]^{\mathsf{M}} & : A^M[\mathsf{id}^{\mathsf{M}}]^{\mathsf{M}} \equiv^{[\mathsf{id}]} A^M \\
\quad \circ\circ^{\mathsf{M}} & : (\sigma^M \circ^{\mathsf{M}} \nu^M) \circ^{\mathsf{M}} \delta^M \equiv^{\circ\circ} \sigma^M \circ^{\mathsf{M}} (\nu^M \circ^{\mathsf{M}} \delta^M)
\end{array}
$$

$$\begin{array}{ll}
\mathsf{id}{\circ}^{\mathsf{M}} & : \mathsf{id}^{\mathsf{M}} \circ^{\mathsf{M}} \sigma^{M} \equiv^{\mathsf{id}\circ} \sigma^{M} \\
{\circ}\mathsf{id}^{\mathsf{M}} & : \sigma^{M} \circ^{\mathsf{M}} \mathsf{id}^{\mathsf{M}} \equiv^{\circ\mathsf{id}} \sigma^{M} \\
\epsilon\eta^{\mathsf{M}} & : \{\sigma^{M} : \mathsf{Tms}^{\mathsf{M}}\, \Gamma^{M} \cdot^{\mathsf{M}}\ \sigma\} \to \sigma^{M} \equiv^{\epsilon\eta} \epsilon^{\mathsf{M}} \\
\pi_1\beta^{\mathsf{M}} & : {\pi_1}^{\mathsf{M}}\, (\sigma^{M},^{\mathsf{M}} t^{M}) \equiv^{\pi_1\beta} \sigma^{M} \\
\pi\eta^{\mathsf{M}} & : ({\pi_1}^{\mathsf{M}}\, \sigma^{M},^{\mathsf{M}} {\pi_2}^{\mathsf{M}}\, \sigma^{M}) \equiv^{\pi\eta} \sigma^{M} \\
,\circ^{\mathsf{M}} & : (\nu^{M},^{\mathsf{M}} t^{M}) \circ^{\mathsf{M}} \sigma^{M} \equiv^{,\circ} (\nu^{M} \circ^{\mathsf{M}} \sigma^{M}),^{\mathsf{M}} ({}_{[][]^{\mathsf{M}}*} t^{M}[\sigma^{M}]^{\mathsf{M}}) \\
\pi_2\beta^{\mathsf{M}} & : {\pi_2}^{\mathsf{M}}\, (\sigma^{M},^{\mathsf{M}} t^{M}) \equiv^{\mathsf{ap}(\mathsf{Tm}^{\mathsf{M}}\, \Gamma^{M}\ A^{M}[-]^{\mathsf{M}} -)\ \pi_1\beta^{\mathsf{M}}\ \pi_2\beta} t^{M}
\end{array}$$

The motives are indexed by the types constituting the QIIT. The methods for point constructors are elements of the motives at the corresponding constructors. The equalities are equalities between the corresponding semantic components over the syntactic equalities. The situation is more subtle in the case of the last equation where the original syntactic equality depends on another syntactic equality. The equality in the syntax is $\pi_2\beta : \pi_2\,(\sigma, t) \equiv^{\pi_1\beta} t$. The types of the two sides are $\pi_2\,(\sigma, t) : \mathsf{Tm}\,\Gamma\, A[\pi_1\,(\sigma, t)]$ and $t : \mathsf{Tm}\,\Gamma\, A[\sigma]$ and they can be shown equal by $\mathsf{ap}\,(\mathsf{Tm}\,\Gamma\, A[-])\,\pi_1\beta$. In the semantics we have ${\pi_2}^{\mathsf{M}}\,(\sigma^{M},^{\mathsf{M}} t^{M}) : \mathsf{Tm}^{\mathsf{M}}\,\Gamma^{M}\, A^{M}[\pi_1^{\mathsf{M}}\,(\sigma^{M},^{\mathsf{M}} t^{M})]\,(\pi_2\,(\sigma, t))$ and $t^{M} : \mathsf{Tm}^{\mathsf{M}}\,\Gamma^{M}\, A^{M}[\sigma^{M}]^{M}\, t$. To show that these types are equal we use the congruence property of $\mathsf{Tm}^{\mathsf{M}}\,\Gamma^{M}\, A^{M}[-]^{\mathsf{M}} -$. Firstly we show equality of $\pi_1^{\mathsf{M}}\,(\sigma^{M},^{\mathsf{M}} t^{M})$ and σ^{M} by $\pi_1\beta^{\mathsf{M}}$. Then we show the equality of $\pi_2\,(\sigma, t)$ and t by $\pi_2\beta$.

We state the type of the eliminators. The computation rules are the same as those for the recursor. We don't list the motives and methods for the dependent function space, U and El, they can be derived using the method in section 4.3.

$$\begin{array}{l}
\mathsf{module\,Elim}\,(m : \mathsf{Model}) \\
\quad \mathsf{open\,Model}\, m \\
\quad \begin{array}{lll}
\mathsf{ElimCon} & : (\Gamma : \mathsf{Con}) & \to \mathsf{Con}^{\mathsf{M}}\,\Gamma \\
\mathsf{ElimTy} & : (A : \mathsf{Ty}\,\Gamma) & \to \mathsf{Ty}^{\mathsf{M}}\,(\mathsf{ElimCon}\,\Gamma)\, A \\
\mathsf{ElimTms} & : (\sigma : \mathsf{Tms}\,\Gamma\,\Delta) & \to \mathsf{Tms}^{\mathsf{M}}\,(\mathsf{ElimCon}\,\Gamma)\,(\mathsf{ElimCon}\,\Delta)\,\sigma \\
\mathsf{ElimTm} & : (t : \mathsf{Tm}\,\Gamma\, A) & \to \mathsf{Tm}^{\mathsf{M}}\,(\mathsf{ElimCon}\,\Gamma)\,(\mathsf{ElimTy}\, A)\, t
\end{array}
\end{array}$$

3.5.1 Normalisation of types and injectivity of Π

In this section we show how to prove injectivity of the type constructor Π using the eliminator. For this, we will first define normal types (types which are either U, El or Π, but not substituted types), then define normalisation of types using the eliminator and then finally show injectivity.

Normal types are given by the following indexed inductive type[1] which is defined mutually with the embedding back into types. We need this mutual definition because we have to go back to types when using context extension in the second argument of the normal type constructor Π. Note that we use overloaded constructor names.

$$\begin{aligned}
&\mathsf{data}\,\mathsf{NTy} && : (\Gamma : \mathsf{Con}) \to \mathsf{Set} \\
&\ulcorner - \urcorner && : \mathsf{NTy}\,\Gamma \to \mathsf{Ty}\,\Gamma \\
&\mathsf{data}\,\mathsf{NTy} && \\
&\quad \mathsf{U} && : \mathsf{NTy}\,\Gamma \\
&\quad \mathsf{El} && : \mathsf{Tm}\,\Gamma\,\mathsf{U} \to \mathsf{NTy}\,\Gamma \\
&\quad \Pi && : (A : \mathsf{NTy}\,\Gamma) \to \mathsf{NTy}\,(\Gamma, \ulcorner A \urcorner) \to \mathsf{NTy}\,\Gamma \\
&\ulcorner \Pi\,A\,B \urcorner && := \Pi\,\ulcorner A \urcorner\,\ulcorner B \urcorner \\
&\ulcorner \mathsf{U} \urcorner && := \mathsf{U} \\
&\ulcorner \mathsf{El}\,\hat{A} \urcorner && := \mathsf{El}\,\hat{A}
\end{aligned}$$

Substitution of normal types can be defined by ignoring the substitution for U, pushing it down to the term for El and pushing it down recursively for Π. We need a mutual definition with a lemma saying that the embedding is compatible with substitution. As NTy doesn't have equality constructors, we use pattern matching notation when defining these functions.

$$\begin{aligned}
&-[-] && : \mathsf{NTy}\,\Theta \to \mathsf{Tms}\,\Gamma\,\Theta \to \mathsf{NTy}\,\Gamma \\
&\ulcorner [] \urcorner && : (A : \mathsf{NTy}\,\Theta)(\sigma : \mathsf{Tms}\,\Gamma\,\Theta) \to \ulcorner A \urcorner[\sigma] \equiv \ulcorner A[\sigma] \urcorner \\
&(\Pi\,A\,B)[\sigma] && := \Pi\,(A[\sigma])\,(B[_{(\ulcorner [] \urcorner\,A\,\sigma)_*}\sigma \uparrow A]) \\
&\mathsf{U}[\sigma] && := \mathsf{U} \\
&(\mathsf{El}\,\hat{A})[\sigma] && := \mathsf{El}\,(\hat{A}[\sigma])
\end{aligned}$$

[1] It is given as an inductive recursive definition which can be translated into an indexed inductive type, see section 2.2.

$$
\begin{aligned}
&\ulcorner[]\urcorner\,(\Pi\,A\,B)\,\sigma := \Pi[] \mathbin{\scriptstyle\blacksquare} \mathsf{ap}\Pi\,\big(\ulcorner[]\urcorner\,A\,\sigma\big)\,\big(\ulcorner[]\urcorner\,B\,(\sigma\uparrow A)\big)\\
&\ulcorner[]\urcorner\,\mathsf{U}\,\sigma \qquad\quad\; := \mathsf{U}[]\\
&\ulcorner[]\urcorner\,(\mathsf{El}\,\hat{A})\,\sigma \quad\;\; := \mathsf{El}[]
\end{aligned}
$$

When defining substitution of Π, we need to use $\ulcorner[]\urcorner$ to transport the lifted substitution $\sigma\uparrow A$ to the expected type. The lemma is proved using the substitution laws of the syntax and recursion in the case of Π. $\mathsf{ap}\Pi$ denotes the congruence rule for Π, its type is $(p_A : A \equiv A') \to B \equiv^{p_A} B' \to \Pi\,A\,B \equiv \Pi\,A'\,B'$.

By induction on normal types, we prove the following two lemmas as well.

$$
\begin{aligned}
&[\mathsf{id}] : (A : \mathsf{NTy}\,\Gamma) \to A[\mathsf{id}] \equiv A\\
&[][] \;\, : (A : \mathsf{NTy}\,\Gamma).\forall\sigma\,\nu.A[\sigma][\nu] \equiv A[\sigma\circ\nu]
\end{aligned}
$$

Now we can define the model of normal types using the following motives for the eliminator.

$$
\begin{aligned}
&\mathsf{Con}^{\mathsf{M}}\,\_ \qquad\;\; := \top\\
&\mathsf{Ty}^{\mathsf{M}}\,\{\Gamma\}\,\_\,A := \Sigma(A' : \mathsf{NTy}\,\Gamma).A \equiv \ulcorner A'\urcorner\\
&\mathsf{Tms}^{\mathsf{M}}\,\_\,\_\,\_ \quad := \top\\
&\mathsf{Tm}^{\mathsf{M}}\,\_\,\_\,\_ \quad\; := \top
\end{aligned}
$$

That is, the eliminator will map a type to a normal type and a proof that the embedding of the normal type is equal to the original type. Contexts, substitutions and terms are mapped to the trivial type. Hence, the methods for contexts, substitutions and terms will be all trivial and the equality methods for them can be proven by refl.

The methods for types are given as follows.

$$
\begin{aligned}
&-[-]^{\mathsf{M}}\,{}_A\,(A', p_A)\,{}_\sigma\,\mathsf{tt} \qquad := \big(A'[\sigma], p_A \mathbin{\scriptstyle\blacksquare} \ulcorner[]\urcorner\,A'\,\sigma\big)\\
&\mathsf{U}^{\mathsf{M}} \qquad\qquad\qquad\qquad\quad := (\mathsf{U}, \mathsf{refl})\\
&\mathsf{El}^{\mathsf{M}}\,{}_{\hat{A}}\,\mathsf{tt} \qquad\qquad\qquad := (\mathsf{El}\,\hat{A}, \mathsf{refl})\\
&\Pi^{\mathsf{M}}\,{}_A\,(A', p_A)\,{}_B\,(B', p_B) := \big(\Pi\,A'\,({}_{p_A *}\,B'), \mathsf{ap}\Pi\,p_A\,p_B\big)
\end{aligned}
$$

$-[-]^{\mathsf{M}}$ receives a type A as an implicit argument, a normal type A' and a proof p_A that they are equal, a substitution σ as an implicit argument, and the semantic version of the substitution which does not carry information. We use the above defined $-[-]$ for substituting A' and we need the concatenation of the equalities p_A and $\ulcorner[]\urcorner$ to provide the equality $\ulcorner A'[\sigma]\urcorner \equiv \ulcorner A[\sigma]\urcorner$. Mapping U and $\mathsf{El}\,\hat{A}$ to normal types is trivial, while in the case of $\Pi\,A\,B$ we use the recursive results A' and B' to construct $\Pi\,A\,B$ and similarly, we use p_A and p_B to construct the equality.

When proving the equality methods $[\mathsf{id}]^{\mathsf{M}}, [][]^{\mathsf{M}}, \mathsf{U}[]^{\mathsf{M}}, \mathsf{El}[]^{\mathsf{M}}$ and $\Pi[]^{\mathsf{M}}$, it is enough to show that the first components of the pairs (the normal types) are equal, the p_A proofs will be equal by K. The equality methods $[\mathsf{id}]^{\mathsf{M}}$ and $[][]^{\mathsf{M}}$ are given by the above lemmas $[\mathsf{id}]$ and $[][]$. The semantic counterparts of the substitution laws $\mathsf{U}[]$ and $\mathsf{El}[]$ are trivial, while $\Pi[]^{\mathsf{M}}$ is given by straightforward induction.

Using the eliminator, we define normalisation of types as follows.

$$\mathsf{norm}\,(A : \mathsf{Ty}\,\Gamma) : \mathsf{NTy}\,\Gamma := \mathsf{proj}_1\,(\mathsf{ElimTy}\,A)$$

We can also show completeness and stability of normalisation. (See section 5.1 for the nomenclature of the properties of normalisation.)

$$\begin{aligned}
&\mathsf{compl}\,(A : \mathsf{Ty}\,\Gamma) : A \equiv \ulcorner\mathsf{norm}\,A\urcorner := \mathsf{proj}_2\,(\mathsf{ElimTy}\,A)\\
&\mathsf{stab}\,(A' : \mathsf{NTy}\,\Gamma) : A' \equiv \mathsf{norm}\,\ulcorner A\urcorner
\end{aligned}$$

Stability is proven by a straightforward induction on normal types.

Injectivity of Π^{NTy} is proven as follows: given a type A, first we define a family $\mathsf{P_A}$ over normal types which maps function types to equality between A and the domain and every other type to $\bot$ (this choice does not matter). Note that in the case of Ty we wouldn't be able to define the same family because it does not respect the equality $\Pi[]$. As a second step, we use J and this family to prove injectivity.

$$\begin{aligned}
&\mathsf{P_A} : \mathsf{NTy}\,\Gamma \to \mathsf{Set}\\
&\mathsf{P_A}\,(\Pi\,A_1\,B_1) := A \equiv A_1\\
&\mathsf{P_A}\,X := \bot\\
&\mathsf{inj}\Pi^{\mathsf{NTy}} : \Pi^{\mathsf{NTy}}\,A\,B \equiv \Pi^{\mathsf{NTy}}\,A'\,B' \to A \equiv A'\\
&\qquad := \mathsf{J}_{(\mathsf{NTy}\,\Gamma)\;(\Pi^{\mathsf{NTy}}\,A\,B)}\,\big(\lambda(X : \mathsf{NTy}\,\Gamma)\,(q : \Pi^{\mathsf{NTy}}\,A\,B \equiv X).\mathsf{P_A}\,X\big)\,\mathsf{refl}
\end{aligned}$$

$\mathsf{P}_A\,(\Pi\,A\,B) = (A \equiv A)$, hence reflexivity is enough to provide the single method of the eliminator J.

We put together these pieces to prove injectivity of Π^{Ty} in the diagram in figure 3.3. We start with a proof $p : \Pi^{\mathsf{Ty}}\,A\,B \equiv \Pi^{\mathsf{Ty}}\,A'\,B'$, then use completeness to get a proof $q : \ulcorner\mathsf{norm}\,(\Pi^{\mathsf{Ty}}\,A\,B)\urcorner \equiv \ulcorner\mathsf{norm}\,(\Pi^{\mathsf{Ty}}\,A'\,B')\urcorner$. Applying norm to both sides and using stability we get $r : \mathsf{norm}\,(\Pi^{\mathsf{Ty}}\,A\,B) \equiv \mathsf{norm}\,(\Pi^{\mathsf{Ty}}\,A'\,B')$. The type of r reduces to $\Pi^{\mathsf{NTy}}\,(\mathsf{norm}\,A)\,(\mathsf{norm}\,B) \equiv \Pi^{\mathsf{NTy}}\,(\mathsf{norm}\,A')\,(\mathsf{norm}\,B')$ and now we can apply the injectivity of normal Π to get that $\mathsf{norm}\,A \equiv \mathsf{norm}\,A'$. As a last step we apply $\ulcorner - \urcorner$ to both sides of this equality and use completeness on A and A' to obtain $A \equiv A'$.

$$\begin{tikzcd}
\Pi^{\mathsf{Ty}}\,A\,B \arrow[r, dash, "p"] \arrow[d, dash, "\mathsf{compl}\,(\Pi\,A\,B)"'] & \Pi^{\mathsf{Ty}}\,A'\,B' \arrow[d, dash, "\mathsf{compl}\,(\Pi\,A'\,B')"] \\
\ulcorner\mathsf{norm}\,(\Pi^{\mathsf{Ty}}\,A\,B)\urcorner \arrow[r, dash, dashed, "q"] & \ulcorner\mathsf{norm}\,(\Pi^{\mathsf{Ty}}\,A'\,B')\urcorner
\end{tikzcd}$$

$$\begin{tikzcd}
\mathsf{norm}\,\ulcorner\mathsf{norm}\,(\Pi^{\mathsf{Ty}}\,A\,B)\urcorner \arrow[r, dash, "\mathsf{ap}\,\mathsf{norm}\,q"] \arrow[d, dash, "\mathsf{stab}\,\big(\mathsf{norm}\,(\Pi^{\mathsf{Ty}}\,A\,B)\big)"'] & \mathsf{norm}\,\ulcorner\mathsf{norm}\,(\Pi^{\mathsf{Ty}}\,A'\,B')\urcorner \arrow[d, dash, "\mathsf{stab}\,\big(\mathsf{norm}\,(\Pi^{\mathsf{Ty}}\,A'\,B')\big)"] \\
\mathsf{norm}\,(\Pi^{\mathsf{Ty}}\,A\,B) \arrow[d, equal] & \mathsf{norm}\,(\Pi^{\mathsf{Ty}}\,A'\,B') \arrow[d, equal] \\
\Pi^{\mathsf{NTy}}\,(\mathsf{norm}\,A)\,(\mathsf{norm}\,B) \arrow[r, dash, dashed, "r"] & \Pi^{\mathsf{NTy}}\,(\mathsf{norm}\,A')\,(\mathsf{norm}\,B')
\end{tikzcd}$$

$$\begin{tikzcd}
\mathsf{norm}\,A \arrow[r, dash, "\mathsf{inj}\Pi^{\mathsf{NTy}}\,r"] & \mathsf{norm}\,A'
\end{tikzcd}$$

$$\begin{tikzcd}
\ulcorner\mathsf{norm}\,A\urcorner \arrow[r, dash, "\mathsf{ap}\,\ulcorner - \urcorner\,\big(\mathsf{inj}\Pi^{\mathsf{NTy}}\,r\big)"] \arrow[d, dash, "\mathsf{compl}\,A"'] & \ulcorner\mathsf{norm}\,A'\urcorner \arrow[d, dash, "\mathsf{compl}\,A'"] \\
A \arrow[r, dash, dashed, "\mathsf{inj}\Pi^{\mathsf{Ty}}\,p"] & A'
\end{tikzcd}$$

Figure 3.3: Proof of injectivity of Π^{Ty}. The dashed lines are given by the fillers of the squares. The double lines are definitional equalities.

3.6 The standard model

In the standard model every syntactic construct is interpreted by its semantic counterpart — this is also sometimes called the metacircular interpretation. That means we interpret contexts as types, types as dependent types indexed over the interpretation of their context, terms as dependent functions and substitutions as functions. The interpretation of Π is metatheoretic dependent function space, the interpretation of lam is metatheoretic lambda etc. The model is parameterised by the interpretation $\llbracket \mathsf{U} \rrbracket : \mathsf{Set}$ of the base type and the interpretation $\llbracket \mathsf{El} \rrbracket : \llbracket \mathsf{U} \rrbracket \to \mathsf{Set}$ of the base family. The interpretation of the equalities are all refl because the two sides are actually convertible in the metatheory. We use pattern matching notation for readability e.g. we write $\mathsf{Ty}^{\mathsf{M}}\,\llbracket \Gamma \rrbracket := t$ instead of $\mathsf{Ty}^{\mathsf{M}} := \lambda \llbracket \Gamma \rrbracket . t$. Note also that we use the double square brackets as part of the variable names, there is no function $\llbracket - \rrbracket$.

$$
\begin{array}{lrl}
\mathsf{Con}^{\mathsf{M}} & & := \mathsf{Set} \\
\mathsf{Ty}^{\mathsf{M}}\,\llbracket \Gamma \rrbracket & & := \llbracket \Gamma \rrbracket \to \mathsf{Set} \\
\mathsf{Tms}^{\mathsf{M}}\,\llbracket \Gamma \rrbracket\,\llbracket \Delta \rrbracket & & := \llbracket \Gamma \rrbracket \to \llbracket \Delta \rrbracket \\
\mathsf{Tm}^{\mathsf{M}}\,\llbracket \Gamma \rrbracket\,\llbracket A \rrbracket & & := (\alpha : \llbracket \Gamma \rrbracket) \to \llbracket A \rrbracket\,\alpha \\
\cdot^{\mathsf{M}} & & := \top \\
\llbracket \Gamma \rrbracket ,^{\mathsf{M}} \llbracket A \rrbracket & & := \Sigma(\alpha : \llbracket \Gamma \rrbracket).\llbracket A \rrbracket\,\alpha \\
\llbracket A \rrbracket [\llbracket \sigma \rrbracket]^{\mathsf{M}} & \alpha & := \llbracket A \rrbracket\,(\llbracket \sigma \rrbracket\,\alpha) \\
\llbracket \sigma \rrbracket \circ^{\mathsf{M}} \llbracket \nu \rrbracket & \alpha & := \llbracket \sigma \rrbracket\,(\llbracket \nu \rrbracket\,\alpha) \\
\mathsf{id}^{\mathsf{M}} & \alpha & := \alpha \\
\epsilon^{\mathsf{M}} & \_ & := \mathsf{tt} \\
\llbracket \sigma \rrbracket ,^{\mathsf{M}} \llbracket t \rrbracket & \alpha & := (\llbracket \sigma \rrbracket\,\alpha, \llbracket t \rrbracket\,\alpha) \\
{\pi_1}^{\mathsf{M}}\,\llbracket \sigma \rrbracket & \alpha & := \mathsf{proj}_1\,(\llbracket \sigma \rrbracket\,\alpha) \\
\llbracket t \rrbracket [\llbracket \sigma \rrbracket]^{\mathsf{M}} & \alpha & := \llbracket t \rrbracket\,(\llbracket \sigma \rrbracket\,\alpha) \\
{\pi_2}^{\mathsf{M}}\,\llbracket \sigma \rrbracket & \alpha & := \mathsf{proj}_2\,(\llbracket \sigma \rrbracket\,\alpha) \\
[][]^{\mathsf{M}} & \alpha & := \mathsf{refl} \\
\ldots & & \\
\mathsf{U}^{\mathsf{M}} & \_ & := \llbracket \mathsf{U} \rrbracket \\
\mathsf{El}^{\mathsf{M}}\,\llbracket \hat{A} \rrbracket & \alpha & := \llbracket \mathsf{El} \rrbracket\,(\llbracket \hat{A} \rrbracket\,\alpha)
\end{array}
$$

$$
\begin{array}{lll}
\mathsf{U}[]^{\mathsf{M}} & & := \mathsf{refl} \\
\mathsf{El}[]^{\mathsf{M}} & & := \mathsf{refl} \\
\Pi^{\mathsf{M}}\, [\![A]\!]\, [\![B]\!] & \alpha & := (x : [\![A]\!]\, \alpha) \to [\![B]\!]\, (\alpha, x) \\
\Pi[]^{\mathsf{M}} & & := \mathsf{refl} \\
\mathsf{lam}^{\mathsf{M}}\, [\![t]\!] & \alpha & := \lambda x. [\![t]\!]\, (\alpha, x) \\
\mathsf{app}^{\mathsf{M}}\, [\![t]\!] & \alpha & := [\![t]\!]\, (\mathsf{proj}_1\, \alpha)\, (\mathsf{proj}_2\, \alpha) \\
\Pi\beta^{\mathsf{M}} & & := \mathsf{refl} \\
\Pi\eta^{\mathsf{M}} & & := \mathsf{refl} \\
\mathsf{lam}[]^{\mathsf{M}} & & := \mathsf{refl}
\end{array}
$$

A consequence of the standard model is consistency, that is in our case we can show that there is no closed term of type U. We use the standard model where $[\![\mathsf{U}]\!]$ is set to be the empty type $\bot$ (the other parameter $[\![\mathsf{El}]\!]$ can be anything).

$$\mathsf{cons}\, (t : \mathsf{Tm}\ \cdot\ \mathsf{U}) : \bot := \mathsf{RecTm}\, t\, \mathsf{tt}$$

Another proof of consistency using normalisation is given in section 5.11.

It should also be clear that to construct the standard model we need a stronger metatheory than the object theory we are considering. In our case this is given by the presence of an additional universe (here we have to eliminate over Set_1).

If we work in homotopy type theory (HoTT), that is, if we don't have K in our metatheory, our definition of the syntax gives a rather strange theory. Because we have not identified any of the equality constructors we introduced this leads to a very non-standard type theory. That is, we may consider two types which have the same syntactic structure but which at some point use two different derivations to derive the same equality but these cannot be shown to be equal. However, this can be easily remedied by truncating our syntax to be a set, i.e. by introducing additional constructors:

$$
\begin{array}{llll}
\mathsf{setTy} & : (A\, B : \mathsf{Ty}\, \Gamma) & (p\, q : A \equiv B) & \to p \equiv q \\
\mathsf{setTms} & : (\sigma\, \nu : \mathsf{Tms}\, \Gamma\, \Delta) & (p\, q : \sigma \equiv \nu) & \to p \equiv q \\
\mathsf{setTm} & : (t\, u : \mathsf{Tm}\, \Gamma\, A) & (p\, q : t \equiv u) & \to p \equiv q
\end{array}
$$

These force our syntax to be a set in the sense of HoTT, i.e. a type for which uniqueness of identity proofs holds. We don't need to do this for Con because it can be shown to be a set from the assumption that $\mathsf{Ty}\,\Gamma$ is a set for every Γ. However, we now run into a different problem: because the eliminator now requires methods corresponding to setTy, setTms, setTm, we can only eliminate into a type which is itself a set. This means that we cannot even define the standard model because we have to eliminate into Set_1, the type of all small types, which is not a set in the sense of HoTT due to univalence (there may be more that one equality proofs between two types). One way around this is to replace Set_1 by the following inductive-recursive universe, which can be shown to be a set.

$\mathsf{data}\,\mathsf{UU} : \mathsf{Set}$

$\mathsf{EL} : \mathsf{UU} \to \mathsf{Set}$

$\mathsf{data}\,\mathsf{UU}$

$\quad '\Pi' : (\hat{A} : \mathsf{UU}) \to (\mathsf{EL}\,\hat{A} \to \mathsf{UU}) \to \mathsf{UU}$

$\quad '\Sigma' : (\hat{A} : \mathsf{UU}) \to (\mathsf{EL}\,\hat{A} \to \mathsf{UU}) \to \mathsf{UU}$

$\quad '\top' : \mathsf{UU}$

$\mathsf{EL}\,('\Pi'\,\hat{A}\,\hat{B}) := (x : \mathsf{EL}\,\hat{A}) \to \mathsf{EL}\,(\hat{B}\,x)$

$\mathsf{EL}\,('\Sigma'\,\hat{A}\,\hat{B}) := \Sigma(x : \mathsf{EL}\,\hat{A}).\mathsf{EL}\,(\hat{B}\,x)$

$\mathsf{EL}\,'\top' := \top$

Nicolai Kraus raised the question whether it may be possible to give the interpretation of a strict model like the standard model with the truncation even though we do not eliminate into a set. This is motivated by his work on general eliminations for the truncation operator [63]. Following this idea it may be possible to eliminate into Set_1 via an intermediate definition which states all the necessary coherence equations.

Chapter 4

Parametricity

In this chapter we introduce unary and binary parametricity informally, then present the formalisation of the unary case. This is a real-world example of using the syntax we defined in the previous chapter. We express parametricity as a syntactic translation following Bernardy et al [25] so we define a mapping from the syntax to the syntax. We don't prove the identity extension lemma (it is also not covered in [25]). This interpretation could be useful in connection with metaprogramming: using a quoting mechanism it could provide a way of automatically deriving parametricity theorems (free theorems [99]) for functions defined in a dependently typed programming language.

4.1 Introduction

4.1.1 Unary parametricity

We introduce unary parametricity through the following example using the theory of figure 3.1 extended with an identity type $- \equiv -$.

$$A : \mathsf{U}, x : A \vdash t : A$$

The above t can be viewed as a program of type A which imports a library through an abstract interface: the interface provides a type A and an element of that type x. Our intuition tells us that as we don't know anything else about the type A, the only way to construct t is to say that $t := x$. This can be made precise by unary parametricity: for this t, unary parametricity says that if there is a predicate on A and x satisfies this predicate, then t also satisfies the predicate. Now, let's define a predicate $A^M : A \to \mathsf{U}$ by $A^M\, y := (y \equiv x)$. We observe that x satisfies this predicate as $A^M\, x = (x \equiv x)$ and we have $\mathsf{refl} : x \equiv x$. Hence, we get that t satisfies the predicate which is $A^M\, t = (t \equiv x)$.

We can formulate this method in general by an operation $-^{\mathsf{P}}$ which works on contexts, types and terms and has the following typing rules (the second rule will become an instance of the third one). The third rule is called parametricity or abstraction theorem.

$$\frac{\Gamma \vdash}{\Gamma^{\mathsf{P}} \vdash} \qquad \frac{\Gamma \vdash A : \mathsf{U}}{\Gamma^{\mathsf{P}} \vdash A^{\mathsf{P}} : A \to \mathsf{U}} \qquad \frac{\Gamma \vdash t : A}{\Gamma^{\mathsf{P}} \vdash t^{\mathsf{P}} : A^{\mathsf{P}}\, t} \tag{4.1}$$

The context Γ^{P} is Γ extended by witnesses that everything in Γ respects the logical predicate, A^{P} is the logical predicate at the type A, t^{P} is the proof of parametricity: in the extended context t respects the predicate A^{P}.

We define $-^{\mathsf{P}}$ inductively on the structure of contexts and terms as follows (we use the theory of figure 3.1, hence we have variable names, implicit substitutions and universes à la Russel).

$$\begin{aligned}
.^{\mathsf{P}} &= \cdot \\
(\Gamma, x : A)^{\mathsf{P}} &= \Gamma^{\mathsf{P}}, x : A, x^M : A^{\mathsf{P}}\, x \\
\mathsf{U}^{\mathsf{P}} &= \lambda A.A \to \mathsf{U} \\
(\Pi(x : A).B)^{\mathsf{P}} &= \lambda f.\Pi(x : A, x^M : A^{\mathsf{P}}\, x).B^{\mathsf{P}}\, (f\, x) \\
x^{\mathsf{P}} &= x^M \\
(\lambda x.t)^{\mathsf{P}} &= \lambda x\, x^M.t^{\mathsf{P}} \\
(f\, a)^{\mathsf{P}} &= f^{\mathsf{P}}\, a\, a^{\mathsf{P}}
\end{aligned}$$

Contexts are extended pointwise with witnesses of the predicate by adding ^{M} indices to the variable names ($-^M$ is just a way to form new variable names). The predicate for U is predicate space. This makes the second rule of 4.1 an instance of the third one as $\mathsf{U}^{\mathsf{P}}\, A = A \to \mathsf{U}$. Also, note that the definition of U^{P} typechecks: we need $\mathsf{U}^{\mathsf{P}} : \mathsf{U}^{\mathsf{P}}\, \mathsf{U}$ and unfolding the definitions on both sides of the colon we get $\lambda A.A \to \mathsf{U} : \mathsf{U} \to \mathsf{U}$. The logical predicate holds for a function $f : \Pi(x : A).B$ if it preserves the predicate i.e. if it maps elements for which A^{P} holds to elements for which B^{P} holds. Note that B^{P} is interpreted in a context $(\Gamma, x : A)^{\mathsf{P}}$, this is why we need to give the names x and x^M to the variables in the domain of the function. The witness of the predicate for a variable can just be projected out from the context, and we can interpret abstraction and application in a straightforward way.

For the definition of $-^{\mathsf{P}}$ to make sense, we need to check whether the left and right hand sides have convertible types and that our definitions respect the conversion rules. We show that this is the case for the formal version in the next section.

Now we show how to derive the above example using the general rules of parametricity defined above.

$$\frac{\dfrac{A : \mathsf{U}, x : A \vdash t : A}{(A : \mathsf{U}, x : A)^{\mathsf{P}} \vdash t^{\mathsf{P}} : A^{\mathsf{P}}\, t}\ \text{parametricity}}{A : \mathsf{U}, x : A \vdash t^{P}[A^{M} := \lambda y.(y \equiv x), x^{M} := \mathsf{refl}] : t \equiv x}$$

4.1.2 Binary parametricity

Another example of using parametricity is given by the following term. This example is given in the theory of figure 3.1 extended by natural numbers, booleans and identity.

$$A : \mathsf{U}, z : A, s : A \to A \vdash t : A$$

Again, t can be viewed as a program parameterised by a type A, an element z of A and a function $s : A \to A$. The context (the things that t depends on) can be interpreted by the following two substitutions for example.

$$\begin{aligned}
\rho_0 &= (A := \mathbb{N}, \quad z := \mathsf{zero}, s := \mathsf{suc}) \\
\rho_1 &= (A := \mathsf{Bool}, z := \mathsf{true}, s := \mathsf{not})
\end{aligned}$$

That is, we provide the parameters of t in the first case by natural numbers with the Peano constructors for z and s and in the second case we have booleans with true for z and negation for successor. We would like to prove that whatever t is, it is not possible to have $t[\rho_0] = \mathsf{suc}\,(\mathsf{suc}\,\mathsf{zero})$ and $t[\rho_1] = \mathsf{false}$. Binary parametricity for t says that if we have a relation between $\mathbb{N}$ and Bool s.t. $z[\rho_0]$ is related to $z[\rho_1]$ and $s[\rho_0]$ and $s[\rho_1]$ preserve related elements, then $t[\rho_0]$ will be related to $t[\rho_1]$. We define the relation $A^{R} : \mathbb{N} \to \mathsf{Bool} \to \mathsf{Set}$ to be $A^{R}\, x\, b := \mathsf{if}\, b\, \mathsf{then}\, \mathsf{Even}\, x\, \mathsf{else}\, \mathsf{Odd}\, x$. ($\mathsf{Even}$ and Odd are predicates on natural numbers expressing that the number is even and odd, respectively. For the definitions, see section 2.3.) We can show that zero and true are related because zero is even. We can also show that that the successor of an even number is odd and the successor of an odd number is even which makes $s[\rho_0]$ and $s[\rho_1]$ preserve related elements (that is, given $n : \mathbb{N}$ and $b : \mathsf{Bool}$, if $A^R\, n\, b$, then also $A^R\, (s[\rho_0]\, n)\, (s[\rho_1]\, b)$). Now parametricity tells us that $t[\rho_0]$ and $t[\rho_1]$ need to be related by A^R, hence it can't happen that $t[\rho_0]$ is 2 and $t[\rho_1]$ is false.

We need a binary version of $-^{\mathsf{P}}$ in order to formulate this kind of reasoning in general. We define the operation $-^{\mathsf{R}}$, i.e. given a type A, A^{R} is a binary relation. The main technical difference is that we need to define projection substitutions ($\mathsf{0}$ and $\mathsf{1}$) because Γ^{R} will contain two copies of Γ that we have to be able to project out. The second rule below expresses that $\mathsf{0}_\Gamma$ and $\mathsf{1}_\Gamma$ are lists of terms, one for each type in Γ, and the all of these terms make sense in the context Γ^{R}. As in the unary case, the third rule will become the instance of the fourth one.

$$\frac{\Gamma \vdash}{\Gamma^{\mathsf{R}} \vdash} \qquad \frac{\Gamma \vdash}{\Gamma^{\mathsf{R}} \vdash \mathsf{0}_\Gamma, \mathsf{1}_\Gamma : \Gamma}$$

$$\frac{\Gamma \vdash A : \mathsf{U}}{\Gamma^{\mathsf{R}} \vdash A^{\mathsf{R}} : A[\mathsf{0}_\Gamma] \to A[\mathsf{1}_\Gamma] \to \mathsf{U}} \qquad \frac{\Gamma \vdash t : A}{\Gamma^{\mathsf{R}} \vdash t^{\mathsf{R}} : A^{\mathsf{R}}\, (t[\mathsf{0}_\Gamma])\, (t[\mathsf{1}_\Gamma])}$$

On contexts and terms, $-^{\mathsf{R}}$ is defined as follows.

$$\begin{aligned}
.^{\mathsf{R}} &= \cdot \\
(\Gamma, x : A)^{\mathsf{R}} &= \Gamma^{\mathsf{P}}, x^0 : A[\mathsf{0}], x^1 : A[\mathsf{1}], x^M : A^{\mathsf{R}}\, x^0\, x^1 \\
\mathsf{U}^{\mathsf{R}} &= \lambda A^0\, A^1 . A^0 \to A^1 \to \mathsf{U} \\
(\Pi(x : A).B)^{\mathsf{R}} &= \lambda f^0\, f^1 . \Pi(x^0 : A[\mathsf{0}], x^1 : A[\mathsf{1}], x^M : A^{\mathsf{R}}\, x^0\, x^1) \\
&\qquad\qquad .B^{\mathsf{R}}\, (f^0\, x^0)\, (f^1\, x^1) \\
x^{\mathsf{R}} &= x^M \\
(\lambda x.t)^{\mathsf{R}} &= \lambda x^0\, x^1\, x^M . t^{\mathsf{R}} \\
(f\, a)^{\mathsf{R}} &= f^{\mathsf{R}}\, (a[\mathsf{0}])\, (a[\mathsf{1}])\, a^{\mathsf{R}}
\end{aligned}$$

$-^0$, $-^1$ and $-^M$ are again just new ways of forming variable names. The substitutions $\mathsf{0}$ and $\mathsf{1}$ project out the corresponding elements from the context i.e. $i_.$ is the empty substitution and $i_{\Gamma.x:A} = (i_\Gamma, x := x^i)$ for $i \in \mathsf{0}, \mathsf{1}$.

Now we use $-^{\mathsf{R}}$ to derive the second example. We start by defining the substitution σ from the empty context into $(A : \mathsf{U}, z : A, s : A \to A)^{\mathsf{R}}$. Note that we use the definition of Even and Odd from section 2.3.

$$\begin{aligned}\sigma = (&A^0 := \mathbb{N},\ \ A^1 := \mathsf{Bool}, A^M := \lambda x^0\, x^1 .\mathsf{if}\ x^1\ \mathsf{then\ Even}\ x^0\ \mathsf{else\ Odd}\ x^0,\\ &x^0 := \mathsf{zero}, x^1 := \mathsf{true},\ x^M := \mathsf{zeroEven},\\ &s^0 := \mathsf{suc},\ s^1 := \mathsf{not},\ \ s^M := \mathsf{lemmaSuc})\end{aligned}$$

lemmaSuc needs to show that suc and not preserve the relation A^M. We prove it by case distinction on the boolean argument x^1.

$$\begin{aligned}&\mathsf{lemmaSuc} : \Pi(x^0 : \mathbb{N}, x^1 : \mathsf{Bool}, x^M : \mathsf{if}\ x^1\ \mathsf{then\ Even}\ x^0\ \mathsf{else\ Odd}\ x^0)\\ &\qquad\qquad .\mathsf{if\ not}\ x^1\ \mathsf{then\ Even}\ (\mathsf{suc}\ x^0)\ \mathsf{else\ Odd}\ (\mathsf{suc}\ x^0)\\ &\mathsf{lemmaSuc}\ x^0\ \ \mathsf{true}\ (x^M : \mathsf{Even}\ x^0) : \mathsf{Odd}\ (\mathsf{suc}\ x^0) := \mathsf{sucEven}\ x^0\ x^M\\ &\mathsf{lemmaSuc}\ x^0\ \ \mathsf{false}\ (x^M : \mathsf{Odd}\ x^0) : \mathsf{Even}\ (\mathsf{suc}\ x^0) := \mathsf{sucOdd}\ x^0\ x^M\end{aligned}$$

In the case when x^1 is true, the type of x^M computes to saying that x^0 is even and our goal computes to saying that $\mathsf{suc}\, x^0$ is odd and this can be given by the constructor sucEven of the datatype Odd. We have the opposite situation in the other case.

We can now put together the pieces and obtain the following.

$$\dfrac{\dfrac{A : \mathsf{U}, z : A, s : A \to A \vdash t : A}{(A : \mathsf{U}, z : A, s : A \to A)^{\mathsf{R}} \vdash t^{\mathsf{R}} : A^{\mathsf{R}}\, t[0]\, t[1]}\ \text{binary parametricity}}{\cdot \vdash t^{\mathsf{R}}[\sigma] : \mathsf{if}\ t[\rho_1]\ \mathsf{then\ Even}\ t[\rho_0]\ \mathsf{else\ Odd}\ t[\rho_0]}$$

4.2 The logical predicate interpretation formalised

We formalise the unary logical predicate interpretation of the type theory given in chapter 3. That is, we will give an element of the record type DModel defined in section 3.5. Note that we do not use function extensionality anywhere when defining this interpretation.

4.2.1 Motives

First we define the motives of the eliminator following the above typing rules for $-^{\mathsf{P}}$ as intuition.

Contexts will be mapped to the lifted ($-^{\mathsf{P}}$-d) context and a projection substitution which goes from the lifted context to the original one. Because we don't have variable names as in the informal presentation, we need the projection Pr whenever we would like to refer to a variable in the lifted context which was present in the original one.

$$\begin{aligned}\mathsf{Con}^{\mathsf{M}}\,(\Gamma : \mathsf{Con}) : \mathsf{Set} := \mathsf{record}\ \ & |{-}| : \mathsf{Con}\\ & \mathsf{Pr}\ \ : \mathsf{Tms}\,|{-}|\,\Gamma\end{aligned}$$

That is, given the interpretation of a context $\Gamma^M : \mathsf{Con}^{\mathsf{M}}\,\Gamma$, we will get the lifted context $|\Gamma^M|$ and a substitution $\mathsf{Pr}\,\Gamma^M : \mathsf{Tms}\,|\Gamma^{\mathsf{M}}|\,\Gamma$.

In the informal presentation of $-^{\mathsf{P}}$ above the predicate for a type was a function from the type to the universe. We simulate such a function by a type (instead of an element of U) in the lifted context extended by the original type. The original type needs to be substituted by Pr because now we are in the larger context.

$$\mathsf{Ty}^{\mathsf{M}}\,\Gamma^M\,A := \mathsf{Ty}\,(|\Gamma^M|, A[\mathsf{Pr}\,\Gamma^M])$$

The interpretation of a substitution is a substitution between the lifted contexts together with a naturality property.

$$\begin{aligned}\mathsf{Tms}^{\mathsf{M}}\,\Gamma^M\,\Delta^M\,\sigma : \mathsf{Set} :=\ \mathsf{record}\ \ & |{-}| \qquad : \mathsf{Tms}\,|\Gamma^M|\,|\Delta^M|\\ & \mathsf{PrNat} : \mathsf{Pr}\,\Delta^M \circ |{-}| \equiv \sigma \circ \mathsf{Pr}\,\Gamma^M\end{aligned}$$

The following diagram depicts the naturality condition given by $\mathsf{PrNat}\,(\sigma^M : \mathsf{Tms}^{\mathsf{M}}\,\Gamma^M\,\Delta^M\,\sigma)$.

$$\begin{tikzcd}
{|\Gamma^M|} \arrow[r, "\mathsf{Pr}\,\Gamma^M"] \arrow[d, "{|\sigma^M|}"'] & \Gamma \arrow[d, "\sigma"] \\
{|\Delta^M|} \arrow[r, "\mathsf{Pr}\,\Delta^M"] & \Delta
\end{tikzcd}$$

Terms are interpreted as witnesses of the predicate at their types in the lifted context. As the predicate needs an additional element $A[\mathsf{Pr}\,\Gamma^M]$, we provide this by the term $t[\mathsf{Pr}\,\Gamma^M]$.

$$\mathsf{Tm}^{\mathsf{M}}\,\Gamma^M\,A^M\,t := \mathsf{Tm}\,|\Gamma^M|\,\big(A^M[\langle t[\mathsf{Pr}\,\Gamma^M]\rangle]\big)$$

This concludes the definition of the motives.

4.2.2 Methods for the substitution calculus

The empty context is interpreted as the empty context and the projection is the empty substitution. An extended context Γ, A is interpreted as the interpretation of the Γ extended by A (which needs to be substituted by the projection) and further extended by A^M which expresses that the logical relation holds for the A in the context. The projection is given by applying $\uparrow$ on the projection for Γ (this way we get the element of type A) and then weakening as we want to forget about the last element having type A^M.

$$\begin{aligned}
\cdot^{\mathsf{M}} \quad &:= (|{-}| := \cdot, &&\mathsf{Pr} := \epsilon)\\
\Gamma^M,^{\mathsf{M}} A^M &:= \big(|{-}| := |\Gamma^M|, A[\mathsf{Pr}\,\Gamma^M], A^M, &&\mathsf{Pr} := (\mathsf{Pr}\,\Gamma^M \uparrow A) \circ \mathsf{wk}\big)
\end{aligned}$$

To define $-[-]^{\mathsf{M}}$ for types, given a predicate $A^M : \mathsf{Ty}\,(|\Theta^M|, A[\mathsf{Pr}\,\Theta^M])$ and a natural substitution $\sigma^M : \mathsf{Tms}^{\mathsf{M}}\,\Gamma^{\mathsf{M}}\,\Theta^M\,\sigma$ we need to substitute the predicate so that we get an element of $\mathsf{Ty}\,\big(|\Gamma^M|, A[\sigma][\mathsf{Pr}\,\Gamma^M]\big)$. We can apply $\uparrow$ on the substitution $|\sigma^M|$ thereby obtaining

$$|\sigma^M| \uparrow A[\mathsf{Pr}\,\Theta^M] : \mathsf{Tms}\,\big(|\Gamma^{\mathsf{M}}|, A[\mathsf{Pr}\,\Theta^M][|\sigma^M|]\big)\,\big(|\Theta^M|, A[\mathsf{Pr}\,\Theta^M]\big).$$

However the domain of this $\uparrow$-d substitution and the context that we need don't match. We can remedy this using the naturality condition in σ^M with the help of which we can prove

$$[][] \centerdot \mathsf{ap}\,(A[-])\,(\mathsf{PrNat}\,\sigma^M) \centerdot [][]^{-1} : A[\mathsf{Pr}\,\Theta^M][|\sigma^M|] \equiv A[\sigma][\mathsf{Pr}\,\Gamma^M].$$

Transporting along this equality we can define the interpretation of a substituted type as follows.

$$A^M[\sigma^M]^{\mathsf{M}} := A^M\big[_{([][] \centerdot \mathsf{ap}\,(A[-])\,(\mathsf{PrNat}\,\sigma^M) \centerdot [][]^{-1})*}|\sigma^M| \uparrow A[\mathsf{Pr}\,\Theta^M]\big]$$

For defining the remaining methods in DModel, we will use the notation of extensional type theory. In practice this means that we check that everything is well-typed up to some transports but we don't write down the transports, see section 2.2.2. The formalisation however is all done in intensional type theory.

The composition of two lifted substitutions is just given by $- \circ -$, and the naturality condition is given by using the naturality conditions of the two substitutions.

$$
\begin{aligned}
\sigma^M \circ^{\mathsf{M}} \nu^M := \big(|{-}| \quad &:= |\sigma^M| \circ |\nu^M| \\
, \mathsf{PrNat} &:= \mathsf{Pr}\,\Delta^M \circ (|\sigma^M| \circ |\nu^M|) \\
&\qquad\qquad\qquad\qquad (\circ\circ^{-1}) \\
&\equiv (\mathsf{Pr}\,\Delta^M \circ |\sigma^M|) \circ |\nu^M| \\
&\qquad\qquad\qquad\qquad (\mathsf{PrNat}\,\sigma^M) \\
&\equiv (\sigma \circ \mathsf{Pr}\,\Theta^M) \circ |\nu^M| \\
&\qquad\qquad\qquad\qquad (\circ\circ) \\
&\equiv \sigma \circ (\mathsf{Pr}\,\Theta^M \circ |\nu^M|) \\
&\qquad\qquad\qquad\qquad (\mathsf{PrNat}\,\nu^M) \\
&\equiv \sigma \circ (\nu \circ \mathsf{Pr}\,\Gamma^M) \\
&\qquad\qquad\qquad\qquad (\circ\circ^{-1}) \\
&\equiv (\sigma \circ \nu) \circ \mathsf{Pr}\,\Gamma^M\big)
\end{aligned}
$$

Lifting of identity is the identity substitution.

$$
\begin{aligned}
\mathsf{id}^{\mathsf{M}} := \big(|{-}| \quad &:= \mathsf{id} \\
, \mathsf{PrNat} &:= \mathsf{Pr}\,\Gamma^M \circ \mathsf{id} \\
&\qquad\qquad (\circ\mathsf{id}) \\
&\equiv \mathsf{Pr}\,\Gamma^M \\
&\qquad\qquad (\mathsf{id}\circ^{-1}) \\
&\equiv \mathsf{id} \circ \mathsf{Pr}\,\Gamma^M\big)
\end{aligned}
$$

Lifting of the empty substitution is the empty substitution.

$$\begin{aligned}
\epsilon^{\mathsf{M}} :=\Big(|{-}| \quad &:= \epsilon \\
,\mathsf{PrNat} &:= \epsilon \circ \epsilon \\
& \qquad\qquad (\epsilon\eta) \\
&\equiv \epsilon \\
& \qquad\qquad (\epsilon\eta^{-1}) \\
&\equiv \epsilon \circ \mathsf{Pr}\,\Gamma^{M}\Big)
\end{aligned}$$

We prove a lemma about $\uparrow$.

$$\begin{aligned}
\uparrow\circ, :&\quad (\sigma \uparrow A) \circ (\nu, t) \equiv (\sigma \circ \nu, t) \\
\uparrow\circ, :=&(\sigma \uparrow A) \circ (\nu, t) \\
&= (\sigma \circ \pi_1\,\mathsf{id}, \pi_2\,\mathsf{id}) \circ (\nu, t) \\
& \qquad\qquad\qquad\qquad (,\circ) \\
&\equiv \big((\sigma \circ \pi_1\,\mathsf{id}) \circ (\nu, t), (\pi_2\,\mathsf{id})[\nu, t]\big) \\
& \qquad\qquad\qquad\qquad (\circ\circ, \pi_1\circ, \mathsf{id}\circ, \pi_1\beta) \\
&\equiv \big(\sigma \circ \nu, (\pi_2\,\mathsf{id})[\nu, t]\big) \\
& \qquad\qquad\qquad\qquad (\pi_2[], \mathsf{id}\circ, \pi_2\beta) \\
&\equiv (\sigma \circ \nu, t)
\end{aligned}$$

The lifting of a substitution extended by a term is the lifting of the substitution, then the term itself (substituted by the projection) and the lifted term. Naturality can be proven using the lemma $\uparrow\circ$, and naturality of the substitution.

$$\begin{aligned}
\sigma^{M},^{\mathsf{M}} t^{M} :=\Big(|{-}| \quad &:= (|\sigma^{M}|, t[\mathsf{Pr}\,\Gamma^{M}], t^{M}) \\
,\mathsf{PrNat} &:= \mathsf{Pr}\,(\Delta^{M},^{\mathsf{M}} A^{M}) \circ (\sigma^{M}, t[\mathsf{Pr}\,\Gamma^{M}], t^{M}) \\
&= \big((\mathsf{Pr}\,\Delta^{M} \uparrow A) \circ \mathsf{wk}\big) \circ (\sigma^{M}, t[\mathsf{Pr}\,\Gamma^{M}], t^{M}) \\
& \qquad\qquad\qquad\qquad (\circ\circ, \pi_1\circ, \pi_1\beta) \\
&\equiv (\mathsf{Pr}\,\Delta^{M} \uparrow A) \circ (\sigma^{M}, t[\mathsf{Pr}\,\Gamma^{M}]) \\
& \qquad\qquad\qquad\qquad (\uparrow\circ,) \\
&\equiv (\mathsf{Pr}\,\Delta^{M} \circ \sigma^{M}, t[\mathsf{Pr}\,\Gamma^{M}]) \\
& \qquad\qquad\qquad\qquad (\mathsf{PrNat}\,\sigma^{M})
\end{aligned}$$

$$\begin{aligned}
&\equiv (\sigma \circ \mathsf{Pr}\,\Gamma^M, t[\mathsf{Pr}\,\Gamma^M]) \\
&\qquad\qquad\qquad\qquad\qquad\qquad (,\circ^{-1}) \\
&\equiv (\sigma, t) \circ \mathsf{Pr}\,\Gamma^M\Big)
\end{aligned}$$

Lifting of a projection substitution is projection twice as the lifted substitution has double size of the original one. To apply $\mathsf{PrNat}\,\sigma^M$ we need to build the projection $\mathsf{Pr}\,(\Delta^M ,^{\mathsf{M}} A^M)$ which is the codomain of $|\sigma^M|$.

$$\begin{aligned}
\pi_1^{\mathsf{M}}\,\sigma^M := \Big(|{-}| \quad &:= \pi_1\,(\pi_1\,|\sigma^M|) \\
, \mathsf{PrNat} &:= \mathsf{Pr}\,\Delta^M \circ \big(\pi_1\,(\pi_1\,|\sigma^M|)\big) \\
&\qquad\qquad\qquad\qquad\qquad\qquad (\pi_1\circ, \mathsf{id}\circ, \circ\circ) \\
&\equiv \mathsf{Pr}\,\Delta^M \circ \mathsf{wk} \circ \mathsf{wk} \circ |\sigma^M| \\
&\qquad\qquad\qquad\qquad\qquad\qquad (\pi_1\beta^{-1}) \\
&\equiv \pi_1\,\big(\mathsf{Pr}\,\Delta^M \circ \mathsf{wk} \circ \mathsf{wk} \circ |\sigma^M|, \mathsf{vz}[\mathsf{wk}][|\sigma^M|]\big) \\
&\qquad\qquad\qquad\qquad\qquad\qquad (,\circ^{-1}) \\
&\equiv \pi_1\,\big((\mathsf{Pr}\,\Delta^M \circ \mathsf{wk} \circ \mathsf{wk}, \mathsf{vz}[\mathsf{wk}]) \circ |\sigma^M|\big) \\
&\qquad\qquad\qquad\qquad\qquad\qquad (,\circ^{-1}) \\
&\equiv \pi_1\,\big(\mathsf{Pr}\,(\Delta^M ,^{\mathsf{M}} A^M) \circ |\sigma^M|\big) \\
&\qquad\qquad\qquad\qquad\qquad\qquad (\mathsf{PrNat}\,\sigma^M) \\
&\equiv \pi_1\,(\sigma \circ \mathsf{Pr}\,\Gamma^M) \\
&\qquad\qquad\qquad\qquad\qquad\qquad (\pi_1\circ^{-1}) \\
&\equiv \pi_1\,\sigma \circ \mathsf{Pr}\,\Gamma^M\Big)
\end{aligned}$$

The methods for terms are simple but to typecheck them we need naturality of $|\sigma^M|$ in both cases.

$$\begin{aligned}
t^M[\sigma^M]^{\mathsf{M}} &:= t^M[|\sigma^M|] \\
\pi_2^{\mathsf{M}}\,\sigma^M &:= \pi_2\,|\sigma^M|
\end{aligned}$$

We state two lemmas about the operator $\uparrow$, the proofs are tedious but straightforward.

$$\begin{aligned}
\circ\!\uparrow &: (\sigma \circ \nu) \uparrow A \equiv (\sigma \uparrow A) \circ (\nu \uparrow A[\sigma]) \\
\mathsf{id}\!\uparrow &: \mathsf{id} \uparrow A \equiv \mathsf{id}
\end{aligned}$$

We omit some details from the proofs for the equalities, we only show the main steps. For $[][]^{\mathsf{M}}$ the main idea is using naturality and lemma $\circ\!\uparrow$. We omit the argument of Pr and PrNat for readability.

$$\begin{aligned}
[][]^{\mathsf{M}} :=& A^M[\sigma^M]^{\mathsf{M}}[\nu^M]^{\mathsf{M}} \\
=& A^M\big[|\sigma^M| \uparrow A[\mathsf{Pr}]\big]\big[|\nu^M| \uparrow A[\sigma][\mathsf{Pr}]\big] \\
& \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad ([][]) \\
\equiv& A^M\big[(|\sigma^M| \uparrow A[\mathsf{Pr}]) \circ (|\nu^M| \uparrow A[\sigma][\mathsf{Pr}])\big] \\
& \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad (\mathsf{PrNat}^{-1}) \\
\equiv& A^M\big[(|\sigma^M| \uparrow A[\mathsf{Pr}]) \circ (|\nu^M| \uparrow A[\mathsf{Pr} \circ |\sigma^M|])\big] \\
& \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad (\circ\!\uparrow^{-1}) \\
\equiv& A^M\big[(|\sigma^M| \circ |\nu^M|) \uparrow A[\mathsf{Pr}]\big]
\end{aligned}$$

The proof of $[\mathsf{id}]^{\mathsf{M}}$ is based on $[\mathsf{id}]$ and $\mathsf{id}\!\uparrow$.

$$\begin{aligned}
[\mathsf{id}]^{\mathsf{M}} :\;& A^M[\mathsf{id}^M] \\
=& A^M[\mathsf{id} \uparrow A[\mathsf{Pr}]] \\
& \qquad\qquad\qquad (\mathsf{id}\!\uparrow) \\
\equiv& A^M[\mathsf{id}] \\
& \qquad\qquad\qquad ([\mathsf{id}]) \\
\equiv& A^M
\end{aligned}$$

To prove that two elements of $\mathsf{Tms}^{\mathsf{M}}\,\Gamma^M\,\Delta^M\,\sigma$ are equal it is enough to prove that the fields $|{-}|$ are equal because the other fields will be equal by K. The first few methods are straightforward, we just use the syntactic construct for the semantics.

$$\begin{aligned}
\circ\circ^{\mathsf{M}} &: (|\sigma^M| \circ |\nu^M|) \circ |\delta^M| \equiv |\sigma^M| \circ (|\nu^M| \circ |\delta^M|) && := \circ\circ \\
\mathsf{id}\circ^{\mathsf{M}} &: \mathsf{id} \circ |\sigma^M| \equiv |\sigma^M| && := \mathsf{id}\circ \\
\circ\mathsf{id}^{\mathsf{M}} &: |\sigma^M| \circ \mathsf{id} \equiv |\sigma^M| && := \circ\mathsf{id} \\
\epsilon\eta^{\mathsf{M}} &: \{\sigma^M : \mathsf{Tms}^{\mathsf{M}}\,\Gamma^M \;\cdot^{\mathsf{M}}\; \sigma\} \to |\sigma^{\mathsf{M}}| \equiv \epsilon && := \epsilon\eta \\
\pi_1\beta^{\mathsf{M}} &: \pi_1\left(\pi_1\left(|\sigma^M|, t[\mathsf{Pr}\,\Gamma^M], t^M\right)\right) \equiv |\sigma^M| && := \mathsf{ap}\,\pi_1\,\pi_1\beta \centerdot \pi_1\beta
\end{aligned}$$

The case of $\pi\eta^{\mathsf{M}}$ is more tricky as we need naturality to transform the expression into a shape where we can apply $\pi\eta$.

$$
\begin{aligned}
\pi\eta^{\mathsf{M}} :\ & \big(\pi_1\,(\pi_1\,|\sigma^M|), (\pi_2\,\sigma)[\mathsf{Pr}\,\Gamma^M], \pi_2\,|\sigma^M|\big) \\
&&& (\pi_2[]) \\
&\equiv \big(\pi_1\,(\pi_1\,|\sigma^M|), \pi_2\,(\sigma \circ \mathsf{Pr}\,\Gamma^M), \pi_2\,|\sigma^M|\big) \\
&&& (\mathsf{PrNat}\,\sigma^M) \\
&\equiv \big(\pi_1\,(\pi_1\,|\sigma^M|), \pi_2\,(\mathsf{Pr}\,(\Delta^M ,^{\mathsf{M}} A^M) \circ |\sigma^M|), \pi_2\,|\sigma^M|\big) \\
&&& (, \circ, \pi_2\beta) \\
&\equiv \big(\pi_1\,(\pi_1\,|\sigma^M|), \pi_2\,(\pi_1\,|\sigma^M|), \pi_2\,|\sigma^M|\big) \\
&&& (\pi\eta) \\
&\equiv \big(\pi_1\,|\sigma^M|, \pi_2\,|\sigma^M|\big) \\
&&& (\pi\eta) \\
&\equiv |\sigma^M|
\end{aligned}
$$

We have a similar situation for $, \circ^{\mathsf{M}}$, we need to apply $, \circ$ twice and then use naturality.

$$
\begin{aligned}
, \circ^{\mathsf{M}} :\ & |\nu^M ,^{\mathsf{M}} t^M| \circ^{\mathsf{M}} \sigma^M \\
&= (|\nu^M|, t[\mathsf{Pr}], t^M) \circ |\sigma^M| \\
&&& (, \circ) \\
&\equiv (|\nu^M|, t[\mathsf{Pr}]) \circ |\sigma^M|, t^M[|\sigma^M|] \\
&&& (, \circ) \\
&\equiv |\nu^M| \circ |\sigma^M|, t[\mathsf{Pr}][|\sigma^M|], t^M[|\sigma^M|] \\
&&& ([][], \mathsf{PrNat}) \\
&\equiv |\nu^M| \circ |\sigma^M|, t[\sigma][\mathsf{Pr}], t^M[|\sigma^M|] \\
&= (\nu^M \circ^{\mathsf{M}} \sigma^M) ,^{\mathsf{M}} t^M[\sigma^M]^{\mathsf{M}}
\end{aligned}
$$

Lastly, we have to verify the single term equality rule $\pi_1\beta^{\mathsf{M}}$ which is just the usage of $\pi_2\beta$.

$$\pi_2\beta^{\mathsf{M}} : \pi_2\,(|\sigma^M|, t[\mathsf{Pr}], t^M) \equiv t^M := \pi_2\beta$$

We summarize the methods below for reference omitting the equality methods, the naturality conditions and the arguments to Pr, PrNat, $\uparrow$.

$$
\begin{aligned}
|\cdot^{M}| &= \cdot \\
|\Gamma^{M},^{\mathsf{M}} A^{M}| &= |\Gamma^{M}|, A[\mathsf{Pr}], A^{M} \\
A^{M}[\sigma^{M}]^{\mathsf{M}} &= A^{M}[|\sigma^{M}|\uparrow] \\
|\sigma^{M} \circ^{\mathsf{M}} \nu^{M}| &= |\sigma^{M}| \circ |\nu^{M}| \\
|\mathsf{id}^{\mathsf{M}}| &= \mathsf{id} \\
|\epsilon^{\mathsf{M}}| &= \epsilon \\
|(\sigma^{M},^{\mathsf{M}} t^{M})| &= |\sigma^{M}|, t[\mathsf{Pr}], t^{M} \\
|\pi_1^{\mathsf{M}}\, \sigma^{M}| &= \pi_1\, (\pi_1\, |\sigma^{M}|) \\
t^{M}[\sigma^{M}]^{\mathsf{M}} &= t^{M}[|\sigma^{M}|] \\
\pi_2^{\mathsf{M}}\, \sigma^{M} &= \pi_2\, |\sigma^{M}|
\end{aligned}
$$

Having defined all the motives and methods we can now define the $-^{\mathsf{P}}$ operation by the $\mathsf{ElimCon}$, ElimTy, $\mathsf{ElimTms}$, ElimTm functions for contexts, types, substitutions and terms, respectively.

4.2.3 Methods for the base type and base family

As in the case of the standard model (section 3.6), we parameterise the logical predicate model with the interpretations of the base type and family. Because the base type is valid in any context and the family only needs the base type to be in the context, the parameters will be the following.

$$
\begin{aligned}
&\bar{\mathsf{U}}\ : \mathsf{Ty}\,(\cdot, \mathsf{U}) \\
&\bar{\mathsf{El}} : \mathsf{Ty}\,(\cdot, \mathsf{U}, \mathsf{El}\,\mathsf{vz}, \bar{\mathsf{U}}[\mathsf{wk}])
\end{aligned}
$$

We define the methods U^{M} and El^{M} as follows.

$$
\begin{aligned}
&\mathsf{U}^{\mathsf{M}} : \mathsf{Ty}\,(|\Gamma^{M}|, \mathsf{U}) := \bar{\mathsf{U}}[\epsilon, \mathsf{vz}] \\
&\mathsf{El}^{\mathsf{M}}\,\{\hat{A} : \mathsf{Tm}\,\Gamma\,\mathsf{U}\}\,\big(\hat{A}^{M} : \mathsf{Tm}\,|\Gamma^{M}|\,(\bar{\mathsf{U}}[\langle \hat{A}[\mathsf{Pr}\,\Gamma^{M}]\rangle])\big) : \mathsf{Ty}\,(|\Gamma^{M}|, \mathsf{El}\,\hat{A}[\mathsf{Pr}\,\Gamma^{M}]) \\
&\qquad := \bar{\mathsf{El}}\big[\epsilon, \hat{A}[\mathsf{Pr}\,\Gamma^{M}][\mathsf{wk}], \mathsf{vz}, \hat{A}^{M}[\mathsf{wk}]\big]
\end{aligned}
$$

The type $\bar{\mathsf{U}}$ only depends on the last element of the context, so we can ignore the part $|\Gamma^{M}|$ by using the empty substitution ϵ. $\bar{\mathsf{El}}$ needs more components:

the U and $\bar{\mathsf{U}}$ components are given by the $\hat{A}$ and $\hat{A}^M$ arguments and the $\mathsf{El}\,\mathsf{vz}$ component is given by the last element in the context.

In addition, we need to verify the substitution laws. Note that $\mathsf{U}[]^\mathsf{M}$ does not say that U^M is invariant to substitutions, but that it is invariant for $\uparrow$-d substitutions (which don't touch the last element in the context, and indeed, that is the only element on which $\bar{\mathsf{U}}$ depends on).

$$\begin{aligned}
\mathsf{U}[]^\mathsf{M} :\ & \mathsf{U}^\mathsf{M}[\sigma^M]^\mathsf{M} \\
&= \hat{\mathsf{U}}[\epsilon, \mathsf{vz}][|\sigma^M| \uparrow] \\
&\qquad\qquad\qquad\qquad \text{(substitution calculus)} \\
&\equiv \hat{\mathsf{U}}[\epsilon \circ (|\sigma^M| \uparrow), \mathsf{vz}] \\
&\qquad\qquad\qquad\qquad (\epsilon\eta) \\
&\equiv \hat{\mathsf{U}}[\epsilon, \mathsf{vz}] \\
&= \mathsf{U}^\mathsf{M}
\end{aligned}$$

$$\begin{aligned}
\mathsf{El}[]^\mathsf{M} :\ & (\mathsf{El}^\mathsf{M}\,\hat{A}^M)[\sigma^M]^\mathsf{M} \\
&= \bar{\mathsf{El}}\big[\epsilon, \hat{A}[\mathsf{Pr}][\mathsf{wk}], \mathsf{vz}, \hat{A}^M[\mathsf{wk}]\big]\,\big[|\sigma^M| \uparrow\big] \\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \text{(substitution calculus and } \epsilon\eta\text{)} \\
&\equiv \bar{\mathsf{El}}\big[\epsilon, \hat{A}[\mathsf{Pr} \circ |\sigma|^M][\mathsf{wk}], \mathsf{vz}, \hat{A}^M[|\sigma^M|][\mathsf{wk}]\big] \\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad (\mathsf{PrNat}\,\sigma^M) \\
&\equiv \bar{\mathsf{El}}\big[\epsilon, \hat{A}[\sigma \circ \mathsf{Pr}][\mathsf{wk}], \mathsf{vz}, \hat{A}^M[|\sigma^M|][\mathsf{wk}]\big] \\
&= \mathsf{El}^\mathsf{M}\,(\hat{A}^M[\sigma^M]^\mathsf{M})
\end{aligned}$$

4.2.4 Methods for the function space

The predicate holds for a function if the function maps inputs for which the predicate holds to outputs for which a predicate holds. Following this we get the interpretation of function space.

$$\begin{aligned}
\Pi^\mathsf{M}\quad & \big(A^M : \mathsf{Ty}\,(|\Gamma^M|, A[\mathsf{Pr}\,\Gamma^M])\big) \\
& \big(B^M : \mathsf{Ty}\,(|\Gamma^M|, A[\mathsf{Pr}\,\Gamma^M], A^M, B[(\mathsf{Pr}\,\Gamma^M \uparrow A) \circ \mathsf{wk})])\big) \\
& \ : \mathsf{Ty}\,\big(|\Gamma^M|, (\Pi\,A\,B)[\mathsf{Pr}\,\Gamma^M]\big) \\
:= \Pi\ & \big(A[\mathsf{Pr}\,\Gamma^M][\mathsf{wk}]\big) \\
& \Big(\Pi\,\big(A^M\big[\mathsf{wk} \uparrow A[\mathsf{Pr}]\big]\big)\ \big(B^M[\mathsf{wk} \uparrow A[\mathsf{Pr}] \uparrow A^M, \mathsf{vs}\,(\mathsf{vs}\,\mathsf{vz})\$\mathsf{vs}\,\mathsf{vz}]\big)\Big)
\end{aligned}$$

As we are in a context extended by a function type, we need to weaken the element of A. A^M needs the context $\Gamma^M, A[\mathsf{Pr}\,\Gamma^M]$. We can go from $\Gamma^M, (\Pi\, A\, B)[\mathsf{Pr}\,\Gamma^M]$ to Γ^M by wk and then by $\uparrow$ we get a substitution

$$\mathsf{wk} \uparrow A[\mathsf{Pr}\,\Gamma^M] : \mathsf{Tms}\,\big(\Gamma^M, (\Pi\, A\, B)[\mathsf{Pr}\,\Gamma^M], A[\mathsf{Pr}\,\Gamma^M][\mathsf{wk}]\big)\,\big(\Gamma^M, A[\mathsf{Pr}\,\Gamma^M]\big).$$

Similarly, we can interpret B^M by applying $\uparrow$ on wk twice and then providing the element of B by applying the function (De Bruijn index 2) to the element of type A (De Bruijn index 1).

The next method is the substitution law for Π^{M}. We verify it by the following equational reasoning. For readability, we omit the second argument for the operator $\uparrow$ and write numerals for De Bruijn indices. Most of the proof is reasoning with laws of the substitution calculus (abbreviated s.c.). We use the following properties: $\sigma \uparrow\uparrow \equiv (\sigma \circ \mathsf{wk}^2, 1, 0)$, $\sigma \uparrow\uparrow\uparrow \equiv (\sigma \circ \mathsf{wk}^3, 2, 1, 0)$ and $|\sigma^M \uparrow^{\mathsf{M}}| \equiv |\sigma^M| \uparrow\uparrow$ ($\uparrow^{\mathsf{M}}$ was defined with the eliminator: it is the semantic counterpart of $\uparrow$).

$$\begin{aligned}
\Pi[]^{\mathsf{M}} :\ & (\Pi^{\mathsf{M}}\, A^M\, B^M)[\sigma^M]^{\mathsf{M}} \\
& = \Big(\Pi\,(A[\mathsf{Pr} \circ \mathsf{wk}])\,\big(\Pi\,(A^M[\mathsf{wk} \uparrow])\,(B^M[\mathsf{wk} \uparrow\uparrow, 2\$1])\big)\Big)[|\sigma^M| \uparrow] \\
& \hspace{22em} (\Pi[]) \\
& \equiv \Big(\Pi\,(A[\mathsf{Pr} \circ \mathsf{wk}][|\sigma^M| \uparrow])\,\big(\Pi\,(A^M[\mathsf{wk} \uparrow][|\sigma^M| \uparrow\uparrow]) \\
& \hspace{10em} (B^M[\mathsf{wk} \uparrow\uparrow, 2\$1][|\sigma^M| \uparrow\uparrow\uparrow])\big)\Big) \\
& \hspace{22em} (\text{s.c.}) \\
& \equiv \Big(\Pi\,(A[\mathsf{Pr} \circ |\sigma^M|][\mathsf{wk}])\,\big(\Pi\,(A^M[|\sigma^M| \circ \mathsf{wk}^2, 0]) \\
& \hspace{10em} (B^M[|\sigma^M| \circ \mathsf{wk}^3, 1, 0, 2\$1])\big)\Big) \\
& \hspace{22em} (\mathsf{PrNat}) \\
& \equiv \Big(\Pi\,(A[\sigma \circ \mathsf{Pr}][\mathsf{wk}])\,\big(\Pi\,(A^M[|\sigma^M| \circ \mathsf{wk}^2, 0]) \\
& \hspace{10em} (B^M[|\sigma^M| \circ \mathsf{wk}^3, 1, 0, 2\$1])\big)\Big) \\
& \hspace{22em} (\text{s.c.}) \\
& \equiv \Pi\,(A[\sigma][\mathsf{Pr} \circ \mathsf{wk}])\,\big(\Pi\,(A^M[|\sigma^M| \uparrow][\mathsf{wk} \uparrow]) \\
& \hspace{10em} (B^M[|\sigma^M| \circ \mathsf{wk}^3, 2, 1, 0][\mathsf{wk} \uparrow\uparrow, 2\$1])\big) \\
& = \Pi^{\mathsf{M}}(A^M[|\sigma^M| \uparrow])\,(B^M[(|\sigma^M| \circ \mathsf{wk}^2, 1, 0) \uparrow]) \\
& = \Pi^{\mathsf{M}}(A^M[\sigma^M]^{\mathsf{M}})\,(B^M[\sigma^M \uparrow^{\mathsf{M}}]^{\mathsf{M}})
\end{aligned}$$

Abstraction and application are quite simple and they follow the informal presentation closely. We just use the constructor twice as the lifted functions take two arguments.

$$\begin{aligned} \mathsf{lam}^{\mathsf{M}}\, t^M &:= \mathsf{lam}\,(\mathsf{lam}\, t^M) \\ \mathsf{app}^{\mathsf{M}}\, t^M &:= \mathsf{app}\,(\mathsf{app}\, t^M) \end{aligned}$$

The β and η laws are just repeated applications of β and η.

$$\begin{aligned} \Pi\beta^{\mathsf{M}} &: \mathsf{app}\left(\mathsf{app}\left(\mathsf{lam}\,(\mathsf{lam}\, t^M)\right)\right) \equiv t^M := \mathsf{ap}\,\mathsf{app}\,\Pi\beta \mathbin{\centerdot} \Pi\beta \\ \Pi\eta^{\mathsf{M}} &: \mathsf{lam}\left(\mathsf{lam}\left(\mathsf{app}\,(\mathsf{app}\, t^M)\right)\right) \equiv t^M := \mathsf{ap}\,\mathsf{lam}\,\Pi\eta \mathbin{\centerdot} \Pi\eta \end{aligned}$$

Finally, we have to verify the naturality law for abstraction which is again just the repeated usage of lam[].

$$\begin{aligned} \mathsf{lam}[]^{\mathsf{M}} :\ & (\mathsf{lam}^{\mathsf{M}}\, t^M)[\sigma^M]^{\mathsf{M}} \\ &= \mathsf{lam}\,(\mathsf{lam}\, t^M)[|\sigma^M|] \\ &\qquad\qquad\qquad\qquad (\mathsf{lam}[] \text{ twice}) \\ &\equiv \mathsf{lam}\left(\mathsf{lam}\,(t^M[|\sigma^M| \uparrow\uparrow])\right) \\ &= \mathsf{lam}^{\mathsf{M}}\,(t^M[\sigma^M \uparrow^{\mathsf{M}}]^{\mathsf{M}}) \end{aligned}$$

4.3 Deriving the eliminator of a closed QIIT

In this section we show an example of the usefulness of logical predicates. Another example is given in chapter 5 where we prove normalisation with the help of a different version of logical predicates.

We will describe a syntactic method for deriving the eliminator from the type formation rules and constructors of a closed QIIT. The construction is restricted to closed types and does not involve syntactic checks like strict positivity. It only derives the type of the eliminator but does not validate the existence of an element of that type. That is we do not derive the existence of inductive types from parametricity as done e.g. in [21] (assuming impredicativity). Our motivation was that we needed a method to derive the eliminator for the syntax of type theory, which works for quotient inductive inductive types.

The contents of this section are not formalised, we will use the informal type theory of figure 3.1 extended with Σ and identity types. We will use the operations $-^{\mathsf{R}}$ and $-^{\mathsf{P}}$ that we defined in section 4.1.

First we show how to extend the unary lifting operation $-^{\mathsf{P}}$ to Σ and identity types. The binary version $-^{\mathsf{R}}$ can be defined analogously.

$-^{\mathsf{P}}$ on Σ types is defined pointwise.

$$
\begin{aligned}
&(\Sigma(x : A).B)^{\mathsf{P}}\,(w : \Sigma(x : A).B) : \mathsf{U} \\
&\qquad = \Sigma(x^M : A^{\mathsf{P}}\,(\mathsf{proj}_1\,w)).B^{\mathsf{P}}[x \mapsto \mathsf{proj}_1\,w]\,(\mathsf{proj}_2\,w) \\
&(a, b)^{\mathsf{P}} = (a^{\mathsf{P}}, b^{\mathsf{P}}) \\
&(\mathsf{proj}_1\,w)^{\mathsf{P}} = \mathsf{proj}_1\,w^{\mathsf{P}} \\
&(\mathsf{proj}_2\,w)^{\mathsf{P}} = \mathsf{proj}_2\,w^{\mathsf{P}}
\end{aligned}
$$

We use the Paulin-Mohring formulation of identity as given in section 2.2.2. $-^{\mathsf{P}}$ on identity is given as follows.

$$
\begin{aligned}
&(a \equiv b)^{\mathsf{P}}\,(q : a \equiv b) : \mathsf{U} && = a^{\mathsf{P}} \equiv^{q} b^{\mathsf{P}} \\
&\mathsf{refl}^{\mathsf{P}} && : a^{\mathsf{P}} \equiv^{\mathsf{refl}} a^{\mathsf{P}} = \mathsf{refl}
\end{aligned}
$$

The lifting of an equality is defined as an equality of liftings (it depends on the original equality as the two sides have different types, $A^{\mathsf{P}}\,a$ and $A^{\mathsf{P}}\,b$, respectively). The lifting of reflexivity is reflexivity. We need to do more work to lift the eliminator J.

$$
\begin{aligned}
\mathsf{J}^{\mathsf{P}}\ & (A : \mathsf{U})(A^M : A \to \mathsf{U})(a : A)(a^M : A^M\, a) \\
& (Q : \Pi(x : A).a \equiv x \to \mathsf{U}) \\
& (Q^M : \Pi(x : A, x^M : A^M\, x, q : a \equiv x, q^M : a^M \equiv^q x^M).Q\, x\, q \to \mathsf{U}) \\
& (r : Q\, a\, \mathsf{refl})(r^M : Q^M\, a\, a^M\, \mathsf{refl}\, \mathsf{refl}\, r) \\
& (x : A)(x^M : A^M\, x)(q : a \equiv x)(q^M : a^M \equiv^q x^M) \\
:\ & Q^M\, x\, x^M\, q\, q^M\, (\mathsf{J}\, A\, a\, Q\, r\, x\, q) \\
:=\ & \mathsf{J}\, (\Sigma(y : A).A^M\, y) \\
& \quad (a, a^M) \\
& \quad \Big(\lambda c\, s.Q^M\, (\mathsf{proj}_1\, c)\, (\mathsf{proj}_2\, c)\, (\mathsf{proj}_{\equiv 1}\, s)\, (\mathsf{proj}_{\equiv 2}\, s) \\
& \qquad\qquad\qquad \big(\mathsf{J}\, A\, a\, Q\, r\, (\mathsf{proj}_1\, c)\, (\mathsf{proj}_{\equiv 1}\, s)\big)\Big) \\
& \quad r^M \\
& \quad (x, x^M) \\
& \quad (q ,_{\equiv} q^M)
\end{aligned}
$$

To define J^{P}, we use J on the Σ type $\Sigma(y : A).A^M\, y$. We used the following helper functions for constructing equalities. They can be seen as constructors and projections for the equality of Σ types (which can be thus viewed as a Σ of equalities).

$$
\begin{aligned}
- ,_{\equiv} -\ & : \Pi(p : a \equiv a').b \equiv^p b' \to (a, b) \equiv (a', b') \\
\mathsf{proj}_{\equiv 1} & : (a, b) \equiv (a', b') \to a \equiv a' \\
\mathsf{proj}_{\equiv 2} & : \Pi(p : (a, b) \equiv (a', b')).b \equiv^{\mathsf{proj}_{\equiv q}\, p} b'
\end{aligned}
$$

In fact, when defining J^{P} we were cheating a bit: the type we get is not

$$Q^M\, x\, x^M\, q\, q^M\, (\mathsf{J}\, A\, a\, Q\, r\, x\, q)$$

but

$$Q^M\, x\, x^M\, \big(\mathsf{proj}_{\equiv 1}\, (q ,_{\equiv} q^M)\big)\, \big(\mathsf{proj}_{\equiv 2}\, (q ,_{\equiv} q^M)\big)\, \Big(\mathsf{J}\, A\, a\, Q\, r\, x\, \big(\mathsf{proj}_{\equiv 1}\, (q ,_{\equiv} q^M)\big)\Big),$$

hence we need to transport it through the following β rules.

$$\begin{aligned}
&\Sigma\beta_{\equiv 1} : \mathsf{proj}_{\equiv 1}\,(p ,_{\equiv} q) \equiv p\\
&\Sigma\beta_{\equiv 2} : \mathsf{proj}_{\equiv 2}\,(p ,_{\equiv} q) \equiv^{\Sigma\beta_{\equiv 1}} q
\end{aligned}$$

The type formation rules and the constructors of a closed QIIT can be given as a context. By closed we mean that they do not refer to other types (except U and Π). E.g. in section 2.2, $\mathbb{N}$ is closed while Vec is not because it is indexed over natural numbers.

As examples we show how $\mathbb{N}$, the Con-Ty fragment of the syntax of type theory (section 2.3) and the interval I (section 2.4) can be given as contexts.

$$\begin{aligned}
&\mathbb{N} : \mathsf{U}, \mathsf{zero} : \mathbb{N}, \mathsf{suc} : \mathbb{N} \to \mathbb{N}\\
&\mathsf{Con} : \mathsf{U}, \mathsf{Ty} : \mathsf{Con} \to \mathsf{U}, \cdot : \mathsf{Con}, -,- : \Pi(\Gamma : \mathsf{Con}).\mathsf{Ty}\,\Gamma \to \mathsf{Con}\\
&, \mathsf{u} : \Pi(\Gamma : \mathsf{Con}).\mathsf{Ty}\,\Gamma, \pi : \Pi(\Gamma : \mathsf{Con}, A : \mathsf{Ty}\,\Gamma).\mathsf{Ty}\,(\Gamma, A) \to \mathsf{Ty}\,\Gamma\\
&\mathsf{I} : \mathsf{U}, \mathsf{left} : \mathsf{I}, \mathsf{right} : \mathsf{I}, \mathsf{segment} : \mathsf{left} \equiv \mathsf{right}
\end{aligned}$$

A substitution into such a context can be seen as an algebra of the corresponding algebraic structure. An algebra of natural numbers is given e.g. by a set, an element of this set and a function from this set to itself.

Unary logical predicates can be used to derive the motives and methods for the eliminator from the algebra. Given an algebra Δ, Δ^{P} contains twice as many elements as Δ: it contains a copy of Δ and additional elements. These additional elements are the motives and methods for the eliminator. We list these for the above examples.

$$\begin{aligned}
&\mathbb{N}^M : \mathbb{N} \to \mathsf{U}, \mathsf{zero}^M : \mathbb{N}^{\mathsf{M}}\,\mathsf{zero}, \mathsf{suc}^M : \Pi(n : \mathbb{N}, n^M : \mathbb{N}^M\,n).\mathbb{N}^M\,(\mathsf{suc}\,n)\\
&\mathsf{Con}^M : \mathsf{Con} \to \mathsf{U}, \mathsf{Ty}^M : \Pi(\Gamma : \mathsf{Con}, \Gamma^M : \mathsf{Con}^M\,\Gamma).\mathsf{Ty}\,\Gamma \to \mathsf{U}, \cdot^M : \mathsf{Con}^M \cdot\\
&, -,^M - : \Pi(\Gamma : \mathsf{Con}, \Gamma^M : \mathsf{Con}^M\,\Gamma, A : \mathsf{Ty}\,\Gamma).\mathsf{Ty}^M\,\Gamma\,\Gamma^M\,A \to \mathsf{Con}^M\,\Gamma\\
&, \mathsf{u}^M : \Pi(\Gamma : \mathsf{Con}, \Gamma^M : \mathsf{Con}^M\,\Gamma).\mathsf{Ty}^M\,\Gamma\,\Gamma^M\,(\mathsf{u}\,\Gamma)\\
&, \pi^M : \Pi\big(\Gamma : \mathsf{Con}, \Gamma^M : \mathsf{Con}^M\,\Gamma, A : \mathsf{Ty}\,\Gamma, A^M : \mathsf{Ty}^M\,\Gamma\,\Gamma^M\,A\\
&\qquad\qquad , B : \mathsf{Ty}\,(\Gamma, A), B^M : \mathsf{Ty}^M\,(\Gamma, A)\,(\Gamma^M ,^M A^M)\,B\big)\\
&\qquad\quad .\mathsf{Ty}^M\,\Gamma\,\Gamma^M\,(\pi\,\Gamma\,A\,B)\\
&\mathsf{I}^M : \mathsf{I} \to \mathsf{U}, \mathsf{left}^M : \mathsf{I}^M\,\mathsf{left}, \mathsf{right}^M : \mathsf{I}^M\,\mathsf{right}\\
&, \mathsf{segment}^M : \mathsf{left}^M \equiv^{\mathsf{segment}} \mathsf{right}^M
\end{aligned}$$

We used this method to define the fields of the record DModel in section 3.5.

A notion of morphism between algebras can be derived using binary logical relations. If the context representing the algebra is denoted Δ, we can use the operation $-^{\mathsf{R}}$ to get the context Δ^{R} with three times as many elements. Δ^{R} contains two copies of Δ and witnesses that the two copies are logically related. The witness for the elements of U are relations. In the case of the above examples we have the following relations.

$$\begin{aligned}
&\mathbb{N}^M && : \mathbb{N}^0 \to \mathbb{N}^1 \to \mathsf{U}\\
&\mathsf{Con}^M && : \mathsf{Con}^0 \to \mathsf{Con}^1 \to \mathsf{U}\\
&\mathsf{Ty}^M && : \Pi(\Gamma^0 : \mathsf{Con}^0, \Gamma^1 : \mathsf{Con}^1, \Gamma^M : \mathsf{Con}^M\,\Gamma^0\,\Gamma^1).\mathsf{Ty}^0\,\Gamma^0 \to \mathsf{Ty}^1\,\Gamma^1 \to \mathsf{U}\\
&\mathsf{I}^M && : \mathsf{I}^0 \to \mathsf{I}^1 \to \mathsf{U}
\end{aligned}$$

Note that in the Con-Ty example we have two relations, one for Con and one for Ty. The latter is indexed over a witness of the previous one.

If we replace these relations in the corresponding Δ^{R}s by graphs of a function, the resulting context becomes the context of two copies of Δ and a homomorphism between them. In our examples the functions would have the following types.

$$\begin{aligned}
&f_{\mathbb{N}} && : \mathbb{N}^0 \to \mathbb{N}^1\\
&f_{\mathsf{Con}} && : \mathsf{Con}^0 \to \mathsf{Con}^1\\
&f_{\mathsf{Ty}} && : (\Gamma^0 : \mathsf{Con}^0) \to \mathsf{Ty}^0\,\Gamma^0 \to \mathsf{Ty}^1\,(f_{\mathsf{Con}}\,\mathsf{Ty}^0)\\
&f_{\mathsf{I}} && : \mathsf{I}^0 \to \mathsf{I}^1
\end{aligned}$$

The $\mathbb{N}$ example becomes the following.

$$\begin{aligned}
&\mathbb{N}^0, \mathbb{N}^1 : \mathsf{U}, f_{\mathbb{N}} : \mathbb{N}^0 \to \mathbb{N}^1, \mathsf{zero}^0 : \mathbb{N}^0, \mathsf{zero}^1 : \mathbb{N}^1, \mathsf{zero}^M : f_{\mathbb{N}}\,\mathsf{zero}^0 \equiv \mathsf{zero}^1\\
&, \mathsf{suc}^0 : \mathbb{N}^0 \to \mathbb{N}^0, \mathsf{suc}^1 : \mathbb{N}^1 \to \mathbb{N}^1\\
&, \mathsf{suc}^M : \Pi(n^0 : \mathbb{N}^0, n^1 : \mathbb{N}^1, n^M : f_{\mathbb{N}}\,n^0 \equiv n^1).f_{\mathbb{N}}\,(\mathsf{suc}^0\,n^0) \equiv (\mathsf{suc}^1\,n^1)
\end{aligned}$$

Using a singleton contraction for suc^M (that is, replacing n^1 by $f_{\mathbb{N}}\,n^0$ and removing the equation n^M) we get the usual notion of homomorphism between the corresponding algebras.

$$\begin{aligned}
&\mathbb{N}^0, \mathbb{N}^1 : \mathsf{U}, f_{\mathbb{N}} : \mathbb{N}^0 \to \mathbb{N}^1 \mathsf{zero}^0 : \mathbb{N}^0, \mathsf{zero}^M : (f_{\mathbb{N}}\,\mathsf{zero}^0 \equiv \mathsf{zero}^1) \\
&, \mathsf{suc}^0 : \mathbb{N}^0 \to \mathbb{N}^0, \mathsf{suc}^1 : \mathbb{N}^1 \to \mathbb{N}^1 \\
&, \mathsf{suc}^M : \Pi(n^0 : \mathbb{N}^0).f_{\mathbb{N}}\,(\mathsf{suc}^0\,n^0) \equiv (\mathsf{suc}^1\,(f_{\mathbb{N}}\,n^0))
\end{aligned}$$

The homomorphism for the Con-Ty example that we obtain using $-^{\mathsf{R}}$, replacing the relations by graphs of the function and using singleton contractions is the following. We omit the ^{0} and ^{1} parts.

$$\begin{aligned}
&\cdot^M : f_{\mathsf{Con}}\,\cdot^0 \equiv \cdot^1 \\
&, -,^M - : \Pi(\Gamma^0 : \mathsf{Con}^0, A^0 : \mathsf{Ty}\,\Gamma^0).f_{\mathsf{Con}}\,(\Gamma^0,^0 A^0) \equiv (f_{\mathsf{Con}}\,\Gamma^0,^1 f_{\mathsf{Ty}}\,\Gamma^0\,A^0) \\
&, \mathsf{u}^M : \Pi(\Gamma^0 : \mathsf{Con}^0).f_{\mathsf{Ty}}\,\Gamma^0\,(\mathsf{u}^0\,\Gamma^0) \equiv \mathsf{u}^1\,(f_{\mathsf{Con}}\,\Gamma^0) \\
&, \pi^M : \Pi(\Gamma^0 : \mathsf{Con}^0, A^0 : \mathsf{Ty}^0\,\Gamma^0, B^0 : \mathsf{Ty}^0\,(\Gamma^0,^0 A^0) \\
&\qquad\qquad .f_{\mathsf{Ty}}\,\Gamma^0\,(\pi^0\,\Gamma^0\,A^0\,B^0) \equiv \pi^1\,(f_{\mathsf{Con}}\,\Gamma^0)\,(f_{\mathsf{Ty}}\,\Gamma^0\,A^0)\,(f_{\mathsf{Ty}}\,(\Gamma^0,^0 A^0)\,B^0)
\end{aligned}$$

The homomorphism derived for the interval example includes an equality stating that the two equality proofs are equal. It is given over other two equalities to make it typecheck (in the unary case, one equality was enough). In our setting with K the equality $\mathsf{segment}^M$ is not very interesting.

$$\begin{aligned}
&\mathsf{left}^M : f_{\mathsf{I}}\,\mathsf{left}^0 \equiv \mathsf{left}^1 \\
&, \mathsf{right}^M : f_{\mathsf{I}}\,\mathsf{right}^0 \equiv \mathsf{right}^1 \\
&, \mathsf{segment}^M : \mathsf{left}^M \equiv^{\mathsf{segment}^0, \mathsf{segment}^1} \mathsf{right}^M
\end{aligned}$$

Note that deriving the notion of homomorphism of algebras is the way we obtain the computation rules for the recursor: these computation rules state that the datatype (an element of the algebra) is the initial algebra in the category of algebras, so the ^{0} component is the inductive datatype definition and the ^{1} component is the algebra comprising the motives and methods of the recursor.

Chapter 5

Normalisation by evaluation

In this chapter we prove normalisation for the theory defined in chapter 3. When we view terms in type theory as functional programs, normalisation is the method of running these programs. When we view terms as proofs, normalisation can be seen as proof theoretic cut elimination.

First we specify what we mean by normalisation (section 5.1), then in section 5.2 we sketch the normalisation by evaluation (NBE) proof for simple type theory following [13] and show how our proof relates to it. After defining some categorical preliminaries in section 5.3, as a warmup, we define the presheaf model for type theory in section 5.4. Our normalisation proof does not depend on this construction, we only add it for intuition and as a reference. Then, after giving a high level overview in section 5.5, we spell out the normalisation proof for the dependent case.

5.1 Specifying normalisation

Normalisation can be given the following specification.

The type of normal forms is denoted $\mathsf{Nf}\,\Gamma\,A$ and there is an embedding from it to terms $\ulcorner - \urcorner : \mathsf{Nf}\,\Gamma\,A \to \mathsf{Tm}\,\Gamma\,A$. Normal forms are defined as a usual inductive type (without equality constructors), hence decidability of their equality should be straightforward. They are defined mutually with neutral terms (terms in which an eliminator is applied to a variable). A neutral term (n) is a variable applied to zero or more normal forms. A normal form (v) is either a neutral term or an abstraction of a normal form. Informally, we define them as follows.

$$
\begin{aligned}
&n ::= x \,|\, n\, v\\
&v ::= n \,|\, \lambda x.v
\end{aligned}
$$

Normalisation is given by a function norm which takes a term and returns a normal form. It needs to be an isomorphism:

$$\text{completeness}\circlearrowleft \qquad \mathsf{norm}\downarrow \frac{\mathsf{Tm}\,\Gamma\,A}{\mathsf{Nf}\,\Gamma\,A} \uparrow \ulcorner{-}\urcorner \qquad \curvearrowright\text{stability}$$

If we normalise a term, we obtain a term which is convertible to it: $t \equiv \ulcorner \mathsf{norm}\, t \urcorner$. This is called completeness. The other direction is called stability: $n \equiv \mathsf{norm} \ulcorner n \urcorner$. It expresses that there is no redundancy in the type of normal forms. This property makes it possible to establish properties of the syntax by induction on normal forms.

Soundness, that is, if $t \equiv t'$ then $\mathsf{norm}\, t \equiv \mathsf{norm}\, t'$ is given by congruence of equality (the ap function). The elimination rule for the QIIT of the syntax ensures that every function defined from the syntax respects the equality constructors.

5.2 NBE from simple to dependent types

NBE is one way to implement this specification. It works by a complete model construction (figure 5.1). We define a model of the syntax and hence we get a function from the syntax to the model given by the eliminator. Then we define a quote function which is a map from the model back into the syntax, but it targets normal forms (which can be viewed as a subset of the syntax via the $\ulcorner{-}\urcorner$ operator). In this section, we informally recall the proof of NBE for simple types and then we describe the difficulties which arise when extending it to dependent types.

5.2.1 NBE for simple types

In this section, we summarize the approach of NBE for simple types by [13]. Here the model we choose is the presheaf model over the category of renamings REN where the base type is interpreted as normal forms of the base type.

Presheaf models are proof-relevant versions of Kripke models (possible world

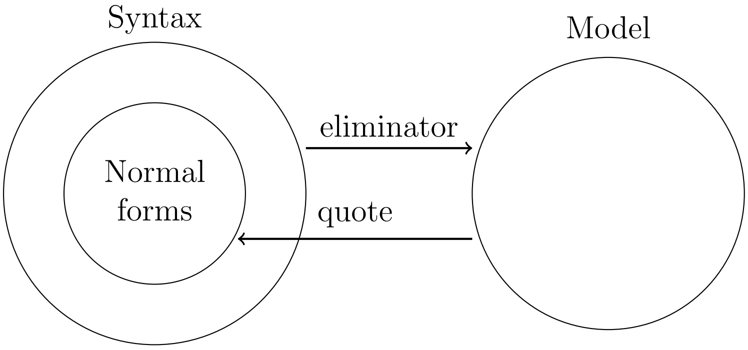

Figure 5.1: Normalisation by evaluation

semantics) for intuitionistic logic: they are parameterised over a category instead of a poset (the normalisation proof using presheaf models can be seen as the proof-relevant version of the completeness proof for intuitionistic logic using Kripke models [9]). A type A is interpreted as a set $[\![A]\!]\,\Psi$ for every object Ψ of the category with maps from the interpretation at one object to the other if there is a corresponding morphism in the category. This is called a presheaf (for this high-level description we omit mentioning the laws that these maps need to satisfy; in section 5.7, we will make these notions precise). Contexts are interpreted as presheaves as well, and a term $t : \mathsf{Tm}\,\Gamma\,A$ is interpreted as a function for each object Ψ from the interpretation of the context at that object $[\![\Gamma]\!]\,\Psi$ to the interpretation of the type at that object $[\![A]\!]\,\Psi$ (this is called a natural transformation). The interpretation of a function type at a given object is a function from the interpretation of the domain to the codomain at any other object from which there is a morphism to the original object (an implication needs to be true in any future world). The set of morphisms from Ω to Ψ are denoted $\mathsf{Hom}(\Omega, \Psi)$.

$$[\![A \to B]\!]\,\Psi = \forall\Omega.(\beta : \mathsf{Hom}(\Omega, \Psi)) \to [\![A]\!]\,\Omega \to [\![B]\!]\,\Omega$$

The interpretation of the base type is a parameter of the presheaf model, it can be set to any presheaf. We will make the notions of presheaves, natural transformations etc. precise in section 5.7, for now we rely on the above intuitive understanding.

We fix the category to be REN (see section 5.9.4 on the possible choices for this category). This category has contexts as objects and renamings (lists of variables) as morphisms. For example, (informally written) from $x : A, y : B, z : C$ we have a morphism $(i \mapsto x, j \mapsto y, k \mapsto x)$ to the context $i : A, j : B, k : A$. We also fix the interpretation of the base type to $\llbracket\iota\rrbracket\,\Psi := \mathsf{Nf}\,\Psi\,\iota$, that is, the base type is interpreted as normal forms. This defines the presheaf model (for the full construction, see section 5.4), and gives an interpretation function from the syntax to the model. E.g. a term $t : \mathsf{Tm}\,\Gamma\,A$ will be mapped to a natural transformation $\llbracket t\rrbracket : \llbracket\Gamma\rrbracket \mathrel{\dot{\to}} \llbracket A\rrbracket$.

After defining the presheaf model, we define another embedding of the syntax into presheaves (independent of the previous presheaf model), which we denote by TM. It can be seen as a variant of Yoneda embedding. A type A can be embedded into the presheaf TM_A by setting $\mathsf{TM}_A\,\Psi = \mathsf{Tm}\,\Psi\,A$ i.e. a type at a given context is interpreted as the set of terms of that type in that context (note that we only have simple types, that is, closed types). Analogously, we can embed a type A into the presheaves of neutral terms NE_A and normal forms NF_A.

Now we define the function quote which goes from the presheaf model to normal forms, c.f. figure 5.1. The quote function for a type A is denoted q_A and is a natural transformation $\llbracket A\rrbracket \mathrel{\dot{\to}} \mathsf{NF}_A$. We define it by induction on the type A. For the base type, it is the identity, as base types are interpreted as normal forms in our presheaf model. For function types, we define[1] it as an abstraction (as we would like a normal form of a function type), then we call quote on the type B at the context extended by A and then we need to produce a semantic value $\llbracket B\rrbracket\,(\Psi, A)$. We do this by calling f: it is able to produce a value at $\llbracket B\rrbracket\,\Omega$ for any Ω from which there is a morphism to Ψ. We choose this morphism to be the weakening renaming which forgets about the last variable. The next argument of f is a semantic value $\llbracket A\rrbracket\,(\Psi, A)$. We can easily produce a term of type $\mathsf{Tm}\,(\Psi, A)\,A$ using the zero De Bruijn index, but we need a semantic version of this. It seems that the only way to produce such a value is to define another function mutually with quote which goes in the other direction, however its domain is not normal forms, but neutral terms. We call this function unquote and denote it $\mathsf{u}_A : \mathsf{NE}_A \mathrel{\dot{\to}} \llbracket A\rrbracket$ for a type A. The of $\mathsf{q}_{A\to B}$ is $\llbracket A \to B\rrbracket \mathrel{\dot{\to}} \mathsf{NF}_{A\to B}$, below we show the unfolded version.

$$
\begin{array}{lllll}
\multicolumn{5}{l}{\mathsf{q}_{A\to B\,\Psi}\,\big(f : \forall\Omega.(\beta : \Omega \to \Psi) \to \llbracket A\rrbracket\,\Omega \to \llbracket B\rrbracket\,\Omega\big) : \mathsf{Nf}\,\Psi\,(A \to B)} \\
:= \mathsf{lam}\,\Big(& \mathsf{q}_{B,(\Psi,A)} & \big(f_{\Psi,A} & \mathsf{wk} & (\mathsf{u}_{A\,(\Psi,A)}\,\mathsf{zero})\big)\Big) \\
 & : \mathsf{Nf}\,(\Psi, A)\,B & : \llbracket B\rrbracket\,(\Psi, A) & : \Psi, A \to \Psi & : \llbracket A\rrbracket\,(\Psi, A)
\end{array}
$$

[1] The definition of quote for function types is related to the fact that Yoneda preserves exponentials.

Unquote is also defined by induction on the type and is identity at the base type and can be defined for the function type as follows. We need to define a semantic function which works for any $\llbracket A \rrbracket \, \Omega$ given a $\beta : \Omega \to \Psi$ morphism. We use unquote of B and application for neutral terms. Application needs a neutral function at context Ω, which we get by renaming the original neutral term with β. To get the normal argument, we quote the semantic element at type A.

$$
\begin{array}{lllll}
\multicolumn{5}{l}{\mathsf{u}_{A \to B\,\Psi}\,(n : \mathsf{Ne}\,\Psi\,(A \to B))\,(\beta : \Omega \to \Psi)(a : \llbracket A \rrbracket\,\Omega) : \llbracket B \rrbracket\,\Omega} \\
:= \mathsf{u}_{B\,\Omega} & \big(\mathsf{app} & (n[\beta]) & & (\mathsf{q}_{A\,\Omega}\,a)\big) \\
\quad : \llbracket B \rrbracket\,\Omega & : \mathsf{Ne}\,\Omega\,B & : \mathsf{Ne}\,\Omega\,(A \to B) & & : \mathsf{Nf}\,\Omega\,A
\end{array}
$$

The types of quote and unquote are summarized by the following diagram of presheaves.

$$\begin{tikzcd}
\mathsf{NE}_A \arrow[r, "\mathsf{u}_A"] & \llbracket A \rrbracket \arrow[r, "\mathsf{q}_A"] & \mathsf{NF}_A
\end{tikzcd}$$

To normalise a term, we also need to define unquote for neutral substitutions (lists of neutral terms). Then we get normalisation by calling unquote on the identity neutral substitution, then interpreting the term at this semantic element and finally quoting.

$$\mathsf{norm}_A\,(t : \mathsf{Tm}\,\Gamma\,A) : \mathsf{Nf}\,\Gamma\,A := \mathsf{q}_{A\,\Gamma}\,\big(\llbracket t \rrbracket\,(\mathsf{u}_\Gamma\,\mathsf{id})\big)$$

Completeness can be proven using a logical relation R between TM and the presheaf model [13]. The logical relation is equality at the base type. We extend quote and unquote to produce witnesses and require a witness of this logical relation, respectively. This is depicted by the commutative diagram in figure 5.2. The commutativity of the right hand triangle gives completeness: starting with a term, a semantic value and a witness that these are related (if the semantic value is the presheaf interpretation of the term then the fundamental theorem of the logical relation says that they are related), we get a normal form, and then if we embed it back into terms, we get a term equal to the one we started with.

Stability can be proven by mutual induction on terms and normal forms.

$$\begin{tikzcd}
\mathsf{NE}_A \arrow[r, "\mathsf{u}'_A"] \arrow[dr, "\ulcorner\_\urcorner"'] & \Sigma\,(\mathsf{TM}_A \times \llbracket A \rrbracket)\,\mathsf{R}_A \arrow[r, "\mathsf{q}'_A"] \arrow[d, "\mathsf{proj}"] & \mathsf{NF}_A \arrow[dl, "\ulcorner\_\urcorner"] \\
& \mathsf{TM}_A &
\end{tikzcd}$$

Figure 5.2: The types of quote and unquote for a type A in NBE for simple types. We use primed notations for the unquote and quote functions to denote that they include the completeness proof. This is a diagram in the category of presheaves.

5.2.2 Extending NBE to dependent types

When we have dependent types, types depend on contexts, hence they are interpreted as families of presheaves in the presheaf model.

$$\begin{aligned}
&\llbracket \Gamma \rrbracket && : \mathsf{REN} \to \mathsf{Set} \\
&\llbracket A : \mathsf{Ty}\,\Gamma \rrbracket && : (\Psi : \mathsf{REN}) \to \llbracket \Gamma \rrbracket\,\Psi \to \mathsf{Set}
\end{aligned}$$

We can declare quote for contexts the same way as for simple types, but the type of quote for types has to be more subtle. One candidate is the following where it depends on quote for contexts.

$$\begin{aligned}
&\mathsf{q}_\Gamma && : (\Psi : \mathsf{REN}) \to \llbracket \Gamma \rrbracket\,\Psi \to \mathsf{Tms}\,\Psi\,\Gamma \\
&\mathsf{q}_{\Gamma\vdash A} && : (\Psi : \mathsf{REN})(\alpha : \llbracket \Gamma \rrbracket\,\Psi) \to \llbracket A \rrbracket_\Psi\,\alpha \to \mathsf{Nf}\,\Psi\,\big(A[\mathsf{q}_{\Gamma,\Psi}\,\alpha]\big)
\end{aligned}$$

The type of unquote needs to depend on quote for contexts.

$$\mathsf{u}_{\Gamma\vdash A} : (\Psi : \mathsf{REN})(\alpha : \llbracket \Gamma \rrbracket\,\Psi) \to \mathsf{Ne}\,\Psi\,\big(A[\mathsf{q}_{\Gamma,\Psi}\,\alpha]\big) \to \llbracket A \rrbracket\,\Psi\,\alpha$$

When we try to define quote and unquote following this specification, we observe that we need some new equations to typecheck our definition. E.g. quote for function types needs that quote after unquote is the identity up to embedding: $\ulcorner - \urcorner \circ \mathsf{q}_A \circ \mathsf{u}_A \equiv \ulcorner - \urcorner$. This is however the consequence of the logical relation between the syntax and the presheaf model: we can read it off figure 5.2 by the commutativity of the diagram: if we embed a neutral term into terms, it is the same as unquoting, then quoting, then embedding.

So let's define quote and unquote mutually with their correctness proofs. It is not very surprising that when moving to dependent types the well-typedness of normalisation depends on completeness. We illustrate the difference by spelling out the type of quote and unquote for contexts using diagrams. Instead of using the following diagram

$$\begin{tikzcd}
\mathsf{NE}_\Gamma \arrow[r, "\mathsf{u}_\Gamma"] & \llbracket \Gamma \rrbracket \arrow[r, "\mathsf{q}_\Gamma"] & \mathsf{NF}_\Gamma,
\end{tikzcd}$$

we try to define quote and unquote as described by the following diagram.

$$\begin{tikzcd}
\mathsf{NE}_\Gamma \arrow[r, "\mathsf{u}_\Gamma"] \arrow[dr, "\ulcorner \_ \urcorner"'] & \Sigma\,(\mathsf{Tms} - \Gamma \times \llbracket \Gamma \rrbracket)\,\mathsf{R}_\Gamma \arrow[r, "\mathsf{q}_\Gamma"] \arrow[d, "\mathsf{proj}"] & \mathsf{NF}_\Gamma \arrow[dl, "\ulcorner \_ \urcorner"] \\
 & \mathsf{TM}_\Gamma &
\end{tikzcd}$$

The type of quote and unquote for types now don't include quote for contexts anymore and become the following (including the commutativity of the triangles above).

$$\begin{aligned}
\mathsf{q}_{\Gamma \vdash A} &: (\Psi : \mathsf{REN})(\rho : \mathsf{Tms}\,\Psi\,\Gamma)(\alpha : \llbracket \Gamma \rrbracket\,\Psi)(p : \mathsf{R}_{\Gamma\,\Psi}\,\rho\,\alpha) \\
&\qquad (t : \mathsf{Tm}\,\Psi\,A[\rho])(v : \llbracket A \rrbracket_\Psi\,\alpha) \to \mathsf{R}_{A\,\Psi\,p}\,t\,v \to \Sigma(n : \mathsf{NF}_A\,\rho).t \equiv \ulcorner n \urcorner \\
\mathsf{u}_{\Gamma \vdash A} &\;\; \Psi\,\rho\,\alpha\,p : (n : \mathsf{Ne}\,\Psi\,A[\rho]) \to \Sigma(v : \llbracket A \rrbracket_\Psi\,\alpha).\mathsf{R}_{A\,\Psi\,p}\,\ulcorner n \urcorner\,v
\end{aligned}$$

Now when we try to define unquote for function types, we need a semantic function as output which needs to work on arbitrary semantic inputs.

$$\begin{aligned}
&\mathsf{proj}_2\Big(\mathsf{u}_{\Gamma \vdash \Pi\,A\,B}\,\Psi\,\rho\,\alpha\,p\,\big(n : \mathsf{Ne}\,\Psi\,(\Pi\,A\,B)[\rho]\big)\Big) \\
&\quad : \forall \Omega.(\beta : \Omega \to \Psi)\big(x : \llbracket A \rrbracket_\Omega\,(\llbracket \Gamma \rrbracket\,\beta\,\alpha)\big) \to \llbracket B \rrbracket_\Omega\,(\llbracket \Gamma \rrbracket\,\beta\,\alpha, x)
\end{aligned}$$

We would like to start defining it by calling unquote on B which needs a proof that x is related to some term. However we only have an arbitrary semantic x, there is no reason for it to be related to some term. It seems that the presheaf model is too big for our purposes.

Our solution is to restrict the presheaf model so that it only contains related semantic elements. We do this by merging the presheaf model and the logical relation into a single proof-relevant logical predicate. We denote the logical predicate at a context Γ by P_Γ. We define normalisation following the diagram in figure 5.3.

$$\begin{tikzcd}
\mathsf{NE}_\Gamma \arrow[r, "\mathsf{u}_\Gamma"] \arrow[dr, "\ulcorner\_\urcorner"'] & \Sigma\,\mathsf{TM}_\Gamma\,\mathsf{P}_\Gamma \arrow[r, "\mathsf{q}_\Gamma"] \arrow[d, "\mathsf{proj}"] & \mathsf{NF}_\Gamma \arrow[dl, "\ulcorner\_\urcorner"] \\
 & \mathsf{TM}_\Gamma &
\end{tikzcd}$$

Figure 5.3: The type of quote and unquote for a context Γ in our proof.

In the presheaf model, the interpretation of the base type was normal forms at the base type and the logical relation at the base type was equality of the term and the normal form. In our case, the logical predicate at the base type will say that there exists a normal form which is equal to the term (this is why it needs to be proof-relevant). This solves the problem mentioned before: now the semantics of a term will be the same term together with a witness of the predicate at that term. Functions in the logical predicate interpretation only need to work on inputs for which the predicate holds.

5.3 Categorical preliminaries

A category $\mathcal{C}$ is given by a type of objects $|\mathcal{C}|$ and given $I, J : |\mathcal{C}|$, a type $\mathcal{C}(I, J)$ which we call the type of morphisms between I and J. A category is equipped with an operation for composing morphisms $- \circ - : \mathcal{C}(J, K) \to \mathcal{C}(I, J) \to \mathcal{C}(I, K)$ and an identity morphism at each object $\mathsf{id}_I : \mathcal{C}(I, I)$. In addition we have the associativity law $(f \circ g) \circ h \equiv f \circ (g \circ h)$ and the identity laws $\mathsf{id} \circ f \equiv f$ and $f \circ \mathsf{id} \equiv f$. Formally, we collect these definitions into a record.

$$\begin{array}{ll}
\multicolumn{2}{l}{\mathsf{record\ Cat}} \\
\quad |-| & : \mathsf{Set} \\
\quad -(-,-) & : |-| \to |-| \to \mathsf{Set} \\
\quad - \circ - & : -(J, K) \to -(I, J) \to -(I, K) \\
\quad \mathsf{id} & : -(I, I) \\
\quad \circ\circ & : (f \circ g) \circ h \equiv f \circ (g \circ h) \\
\quad \mathsf{id}\circ & : \mathsf{id} \circ f \equiv f \\
\quad \circ\mathsf{id} & : f \circ \mathsf{id} \equiv f
\end{array}$$

Just as the definition of category, the following definitions can be formalised using records.

A contravariant presheaf over a category $\mathcal{C}$ is denoted $\Gamma : \mathsf{PSh}\,\mathcal{C}$. It is given by the following data: given $I : |\mathcal{C}|$, a set $\Gamma\,I$, and given $f : \mathcal{C}(J, I)$ a function $\Gamma\,f : \Gamma\,I \to \Gamma\,J$. Moreover, we have $\mathsf{idP}\,\Gamma : \Gamma\,\mathsf{id}\,\alpha \equiv \alpha$ and $\mathsf{compP}\,\Gamma : \Gamma\,(f \circ g)\,\alpha \equiv \Gamma\,g\,(\Gamma\,f\,\alpha)$ for $\alpha : \Gamma\,I$, $f : \mathcal{C}(J, I)$, $g : \mathcal{C}(K, J)$.

A natural transformation between presheaves Γ and Δ is denoted $\sigma : \Gamma \mathrel{\dot{\rightarrow}} \Delta$. It is given by a function $\sigma : \{I : |\mathcal{C}|\} \to \Gamma\,I \to \Delta\,I$ together with the condition $\mathsf{natn}\,\sigma : \Delta\,f\,(\sigma_I\,\alpha) \equiv \sigma_J\,(\Gamma\,f\,\alpha)$ for $\alpha : \Gamma\,I$, $f : \mathcal{C}(J, I)$.

Given $\Gamma : \mathsf{PSh}\,\mathcal{C}$, a family of presheaves over Γ is denoted $A : \mathsf{FamPSh}\,\Gamma$. It can be seen as a "presheaf" depending on another presheaf. It is given by a set for each element of the presheaf, that is, for any $\alpha : \Gamma\,I$ we have a set $A_I\,\alpha$. Moreover, we can use the morphisms in $\mathcal{C}$ to transport elements of this set: given $f : \mathcal{C}(J, I)$, we can go from A at I to A at J. Because the sets also depend on α, we also need to transport α, hence the type of the transport function we have for A is $A_I\,\alpha \to A_J\,(\Gamma\,f\,\alpha)$. We denote it by $A\,f$. In addition, we have the functor laws $\mathsf{idF}\,A : A\,\mathsf{id}\,v \equiv^{\mathsf{idP}} v$ and $\mathsf{compF}\,A : A\,(f \circ g)\,v \equiv^{\mathsf{compP}} A\,g\,(A\,f\,v)$ for $\alpha : \Gamma\,I$, $v : A\,\alpha$, $f : \mathcal{C}(J, I)$, $g : \mathcal{C}(K, J)$.

A family of natural transformations between two families of presheaves $A, B : \mathsf{FamPSh}\,\Gamma$ is given is denoted $\sigma : A \overset{\mathsf{N}}{\rightarrow} B$. It is given by a function

$$\sigma : \{I : |\mathcal{C}|\}\{\alpha : \Gamma\,I\} \to A_I\,\alpha \to B_I\,\alpha$$

together with the condition $B\,f\,(\sigma_{I\,\alpha}\,a) \equiv \sigma_{J\,(\Gamma\,f\,\alpha)}\,(A\,f\,a)$ for $a : A_I\,\alpha$, $\alpha : \Gamma\,I$, $f : \mathcal{C}(J, I)$.

A section[2] from a presheaf Γ to a family of presheaves A over Γ is denoted $t : \Gamma \overset{\mathsf{s}}{\rightarrow} A$. It is given by a function $t : \{I : |\mathcal{C}|\} \to (\alpha : \Gamma\,I) \to A_I\,\alpha$ together with the naturality condition $\mathsf{natS}\,t\,\alpha\,f : A\,f\,(t\,\alpha) \equiv t\,(\Gamma\,f\,\alpha)$ for $f : \mathcal{C}(J, I)$.

Given $\Gamma : \mathsf{PSh}\,\mathcal{C}$ and $A : \mathsf{FamPSh}\,\Gamma$ we can define $\Sigma\,\Gamma\,A : \mathsf{PSh}\,\mathcal{C}$ by $(\Sigma\,\Gamma\,A)\,I := \Sigma(\alpha : \Gamma\,I).A_I\,\alpha$ and $(\Sigma\,\Gamma\,A)\,f\,(\alpha, a) := (\Gamma\,f\,\alpha, A\,f\,a)$.

Given $\sigma : \Gamma \mathrel{\dot{\rightarrow}} \Delta$ and $A : \mathsf{FamPSh}\,\Delta$, we define $A[\sigma] : \mathsf{FamPSh}\,\Gamma$ by $A[\sigma]_I\,\alpha := A_I\,(\sigma_I\,\alpha)$ and $A[\sigma]\,f\,\alpha := {}_{\mathsf{natn}\,\sigma *}(A\,f\,\alpha)$ for $\alpha : \Gamma\,I$ and $f : \mathcal{C}(J, I)$.

The weakening natural transformation $\mathsf{wk} : \Sigma\,\Gamma\,A \mathrel{\dot{\rightarrow}} \Gamma$ is defined by $\mathsf{wk}_I\,(\alpha, a) := \alpha$.

[2] $t : \Gamma \overset{\mathsf{s}}{\rightarrow} A$ is called a section because it can be viewed as a section of the first projection from $\Sigma\,\Gamma\,A$ to Γ but we define it without using the projection.

Lifting of a section $t : \Gamma \overset{\mathsf{s}}{\dot\to} A$ by a family of presheaves $B : \mathsf{FamPSh}\,\Gamma$ is a natural transformation $t \uparrow B : \Sigma\,\Gamma\,B \dot\to \Sigma\,(\Sigma\,\Gamma\,A)\,B[\mathsf{wk}]$. It is defined as $t \uparrow B_I\,(\alpha, b) := (\alpha, t_I\,\alpha, b)$.

5.4 Warmup: presheaf model

As a warmup to the normalisation proof, we define the presheaf model of our type theory given in section 3.2. No later technical constructions depend on this section, but it is helpful for later intuition.

The presheaf model is parameterised over a category $\mathcal{C}$, a presheaf $[\![\mathsf{U}]\!]$ and a family of presheaves over $[\![\mathsf{U}]\!]$ denoted $[\![\mathsf{El}]\!]$. We use the recursor to define the model, the motives are the following.

$$
\begin{aligned}
&\mathsf{Con}^{\mathsf{M}} && := \mathsf{PSh}\,\mathcal{C}\\
&\mathsf{Ty}^{\mathsf{M}}\,[\![\Gamma]\!] && := \mathsf{FamPSh}\,[\![\Gamma]\!]\\
&\mathsf{Tms}^{\mathsf{M}}\,[\![\Delta]\!]\,[\![\Gamma]\!] && := [\![\Delta]\!] \dot\to [\![\Gamma]\!]\\
&\mathsf{Tm}^{\mathsf{M}}\,[\![\Gamma]\!]\,[\![A]\!] && := [\![\Gamma]\!] \overset{\mathsf{s}}{\dot\to} [\![A]\!]
\end{aligned}
$$

Note that here $[\![-]\!]$ is not a function, it is just part of the variable names. The interpretation of a context is a contravariant presheaf over $\mathcal{C}$. The interpretation of a type over Γ is a family of presheaves over the interpretation of Γ. The interpretation of a substitution $\mathsf{Tms}\,\Delta\,\Gamma$ is a natural transformation from the interpretations of Δ to the interpretation of Γ. The interpretation of a term $\mathsf{Tm}\,\Gamma\,A$ is a section from the interpretation of Γ to that of A.

The methods for the recursor interpret the constructors for contexts, types, substitutions and terms including the equality constructors.

We start with the context constructors. These are presheaves, we only spell out the action on objects.

$$
\begin{aligned}
&\cdot^{\mathsf{M}}\ I && := \top\\
&[\![\Gamma]\!] \rhd^{\mathsf{M}} [\![A]\!]\,I && := \Sigma(\alpha : [\![\Gamma]\!]\,I).[\![A]\!]\,\alpha
\end{aligned}
$$

The empty context is the constant unit presheaf, context extension is pointwise.

We list the interpretations of type formers only showing the action on objects.

$$
[\![A]\!][[\![\rho]\!]]^{\mathsf{M}}\,\alpha \quad := [\![A]\!]\,([\![\rho]\!]\,\alpha)
$$

$$
\begin{aligned}
&\mathsf{U}^{\mathsf{M}}{}_{I}\,\alpha && := [\![\mathsf{U}]\!]\,I \\
&\mathsf{El}^{\mathsf{M}}\,[\![\hat{A}]\!]\,\alpha && := [\![\mathsf{El}]\!]\,([\![\hat{A}]\!]\,\alpha) \\
&\Pi^{\mathsf{M}}\,[\![A]\!]\,[\![B]\!]\,\alpha && := \mathsf{ExpPSh}\,[\![A]\!]\,[\![B]\!]
\end{aligned}
$$

Substituted types are interpreted using the interpretation of the substitution on the environment. U and El are interpreted using the parameters of the presheaf model. Note that U^{M} does not depend on the environment. The interpretation of Π is the dependent presheaf exponential which consists of a function map together with a compatibility condition.

$$
\begin{aligned}
&\mathsf{record}\,\mathsf{ExpPSh}\,([\![A]\!] : \mathsf{FamPSh}\,[\![\Gamma]\!])\,([\![B]\!] : \mathsf{FamPSh}\,[\![\Gamma, A]\!]) : \mathsf{Set} \\
&\quad \mathsf{map} : \forall\{J\}(f : \mathcal{C}(J, I))(x : [\![A]\!]_J\,([\![\Gamma]\!]\,f\,\alpha)) \to [\![B]\!]_J\,([\![\Gamma]\!]\,f\,\alpha, x) \\
&\quad \mathsf{comp} : \forall\{f\,g\,x\}.[\![B]\!]\,g\,(\mathsf{map}\,f\,x) \equiv^{\mathsf{compP}^{-1}} \mathsf{map}\,(g \circ f)\,({}_{\mathsf{compP}^{-1}*}[\![A]\!]\,g\,x)
\end{aligned}
$$

The function maps for any morphism (any future world with the Kripke analogy) the interpretation of A at the environment transported along this morphism to the interpretation of B. To state the compatibility condition, on the right hand side we start with $[\![A]\!]\,g\,x : [\![A]\!]\,([\![\Gamma]\!]\,g\,([\![\Gamma]\!]\,f\,\alpha))$, but to apply map, we need to transport this along compP^{-1} to get an element of type $[\![A]\!]\,([\![\Gamma]\!]\,(g \circ f)\,\alpha)$. Now we can apply $\mathsf{map}\,(g \circ f)$ to it.

The interpretations of substitution constructors are listed below omitting the naturality proofs.

$$
\begin{aligned}
&\mathsf{id}^{\mathsf{M}}\,\alpha && := \alpha \\
&([\![\rho]\!] \circ^{\mathsf{M}} [\![\sigma]\!])\,\alpha && := [\![\rho]\!]\,([\![\sigma]\!]\,\alpha) \\
&\epsilon^{\mathsf{M}}\,\alpha && := \mathsf{tt} \\
&([\![\rho]\!],^{\mathsf{M}}[\![t]\!])\,\alpha && := ([\![\rho]\!]\,\alpha, [\![t]\!]\,\alpha) \\
&\pi_1{}^{\mathsf{M}}\,[\![\rho]\!]\,\alpha && := \mathsf{proj}_1\,([\![\rho]\!]\,\alpha)
\end{aligned}
$$

Identity becomes identity, composition composition, the empty substitution is interpreted as the element of the unit type, comprehension is pointwise, the first projection becomes first projection.

The interpretations of term formers are listed below omitting the naturality proofs and the compatibility condition for $\mathsf{lam}^{\mathsf{M}}$.

$$\begin{aligned}
&[\![t]\!][[\![\rho]\!]]^{\mathsf{M}}\,\alpha && := [\![t]\!]\,([\![\rho]\!]\,\alpha)\\
&{\pi_2}^{\mathsf{M}}\,[\![t]\!]\,\alpha && := \mathsf{proj}_2\,([\![\rho]\!]\,\alpha)\\
&\mathsf{map}\,\big(\mathsf{lam}^{\mathsf{M}}\,[\![t]\!]\,(\alpha : [\![\Gamma]\!]\,I)\big) && := \lambda f\,x.[\![t]\!]([\![\Gamma]\!]\,f\,\alpha, x)\\
&\mathsf{app}^{\mathsf{M}}\,[\![t]\!]\,\alpha && := \mathsf{map}\,\big([\![t]\!]\,(\mathsf{proj}_1\,\alpha)\big)\,\mathsf{id}\,(\mathsf{proj}_2\,\alpha)
\end{aligned}$$

We don't list the interpretations of the equality constructors. Interestingly, up to function extensionality and coercions, all equality proofs are reflexivity.

5.5 Overall structure of the proof

In this section, we give a high level sketch of the normalisation proof.

In section 5.6 we define the category of renamings REN: objects are contexts and morphisms are renamings (lists of variables).

In section 5.7 we define the proof-relevant presheaf logical predicate interpretation of the syntax (sometimes this is called Kripke logical predicate interpretation). The interpretation has REN as the base category and two parameters for the interpretations of U and El. This interpretation can be seen as a dependent version of the presheaf model of type theory [58]. For example, a context in the presheaf model is interpreted as a presheaf. Now it is a family of presheaves which depends on a substitution into that context. The interpretations of base types can depend on the actual elements of the base types. The interpretation of substitutions and terms are sometimes called the fundamental theorems.

Note that this logical predicate interpretation is different from the one given in chapter 4: the target of that interpretation was the object theory, while here it is the metatheory. In addition, this interpretation is parameterised over a category (which we fix to be REN). The exact relationship can be probably given using the notion of categorical glueing. We sketch the idea here. Models of type theory can be given as categories with additional structure (e.g. categories with families, see section 3.4). Given two models $\mathcal{C}$ and $\mathcal{D}$ and a morphism f between them, glueing generates a new model $\mathcal{G}$. In this model objects (contexts) are given as triples of a $\Gamma : |\mathcal{C}|$, $\Delta : |\mathcal{D}|$ and a morphism $\mathcal{D}(\Delta, f\,\Gamma)$. We don't spell out the rest of the construction, we refer to [42, 91]. In the case of the internal logical predicate interpretation of chapter 4, $\mathcal{C}$ and $\mathcal{D}$ are both CON (the category of contexts and substitutions) and the morphism is the identity functor. Thus in the glued representation we have two contexts (corresponding to Γ and Γ^{P}) and a morphism $\mathsf{CON}(\Gamma^{\mathsf{P}}, \Gamma)$ corresponding to the projection Pr. In the case of the presheaf logical predicate interpretation, $\mathcal{C}$ is CON, $\mathcal{D}$ is the presheaf model over REN and the morphism is the Yoneda embedding TM given in section 5.7.1. Now in the glued model, the triples constituting a context Γ are equivalent to families of presheaves over TM_Γ (which is how we will define the logical predicate interpretation of a context Γ).

In section 5.8 we define neutral terms and normal forms together with their renamings and embeddings into the syntax ($\ulcorner - \urcorner$). With the help of these, we define the interpretations of U and El which are given as parameters of the presheaf logical predicate. The interpretation of U at a term of type U will be a neutral term of type U which is equal to the term. Now we can interpret any term of the syntax in the logical predicate interpretation. We will denote the interpretation of a term t by P_t.

In section 5.9 we mutually define the natural transformations quote and unquote. We define them by induction on contexts and types so that they have types for contexts as shown in figure 5.3. Quote takes a term and a witness of the logical predicate at that term into a normal term and a proof that the normal term is equal to it. Unquote takes a neutral term into a witness of the predicate at the neutral term.

In section 5.10, we put together the pieces by defining the normalisation function and showing that it is complete and stable. Normalisation and completeness are given by interpreting the term at the identity semantic element and then quoting. Stability is proved by mutual induction on neutral terms and normal forms.

In section 5.11 we make use of normalisation by proving that our type theory is consistent, i.e. there is no term of the base type in the empty context.

5.6 The category of renamings

In this section we define the category REN. Objects in this category are contexts, morphisms are renamings (Vars): lists of De Bruijn variables.

We define Var as the type of typed De Bruijn indices indexed by syntactic types.

$$
\begin{array}{ll}
\mathsf{data\,Var} & : (\Psi : \mathsf{Con}) \to \mathsf{Ty}\,\Psi \to \mathsf{Set} \\
\quad \mathsf{vz} & : \mathsf{Var}\,(\Psi, A)\,(A[\mathsf{wk}]) \\
\quad \mathsf{vs} & : \mathsf{Var}\,\Psi\,A \to \mathsf{Var}\,(\Psi, B)\,(A[\mathsf{wk}])
\end{array}
$$

The constructor vz projects out the last element of the context, vs extends the context, and the type $A : \mathsf{Ty}\,\Psi$ needs to be weakened in both cases because we need to interpret it in Ψ extended by another type. The weakening wk was defined as $\pi_1\,\mathsf{id}$ in section 3.2.1.

We define renamings Vars as lists of variables together with their embeddings into substitutions $\ulcorner - \urcorner$.

$$
\begin{array}{ll}
\ulcorner - \urcorner & : \mathsf{Vars}\,\Omega\,\Psi \to \mathsf{Tms}\,\Omega\,\Psi \\
\mathsf{data\,Vars} & : \mathsf{Con} \to \mathsf{Con} \to \mathsf{Set} \\
\quad \epsilon & : \mathsf{Vars}\,\Psi\,\cdot \\
\quad -,- & : (\beta : \mathsf{Vars}\,\Omega\,\Psi) \to \mathsf{Var}\,\Omega\,A[\ulcorner \beta \urcorner] \to \mathsf{Vars}\,\Omega\,(\Psi, A) \\
\ulcorner - \urcorner & : \mathsf{Var}\,\Psi\,A \to \mathsf{Tm}\,\Psi\,A \\
\ulcorner \mathsf{vz} \urcorner & := \mathsf{vz} \\
\ulcorner \mathsf{vs}\,x \urcorner & := \mathsf{vs}\,\ulcorner x \urcorner \\
\ulcorner \epsilon \urcorner & := \epsilon \\
\ulcorner \beta, x \urcorner & := \ulcorner \beta \urcorner, \ulcorner x \urcorner
\end{array}
$$

Embedding of variables into terms uses the De Bruijn constructors given in section 3.2.1, the overloading of constructor names is justified by this embedding. Embedding of renamings is performed pointwise.

We will use the names Ψ, Ω, Ξ for objects of REN, x, y for variables, β, γ for renamings.

As a first step in defining the categorical structure, we define renaming a variable by induction on the variable. Note that it is not immediate that a renaming into a context Γ, A (as in the case of vz and $\mathsf{vs}\,x$ below) must be of the form β, y and can't be ϵ. We need disjointness of $\cdot$ and Γ, A (section 3.4.1) to show this; for readability however, we will omit these details below.

$$\begin{aligned}
&-[-] &&: \mathsf{Var}\,\Psi\,A \to (\beta : \mathsf{Vars}\,\Omega\,\Psi) \to \mathsf{Var}\,\Omega\,A[\ulcorner\beta\urcorner] \\
&\mathsf{vz}[\beta, y] &&:= y \\
&(\mathsf{vs}\,x)[\beta, y] &&:= x[\beta]
\end{aligned}$$

We prove the following compatibility law by induction on x.

$$\ulcorner[]\urcorner : \ulcorner x\urcorner[\ulcorner\beta\urcorner] \equiv \ulcorner x[\beta]\urcorner$$

Now we are able to define composition of renamings mutually with compatibility.

$$\begin{aligned}
&- \circ - &&: \mathsf{Vars}\,\Xi\,\Psi \to \mathsf{Vars}\,\Omega\,\Xi \to \mathsf{Vars}\,\Omega\,\Psi \\
&\ulcorner\circ\urcorner &&: \ulcorner\beta\urcorner \circ \ulcorner\gamma\urcorner \equiv \ulcorner\beta \circ \gamma\urcorner \\
&\epsilon \circ \gamma &&:= \epsilon \\
&(\beta, x) \circ \gamma &&:= \beta \circ \gamma,_{\ulcorner\circ\urcorner *} x[\gamma]
\end{aligned}$$

To define the identity renaming, we first need weakening of renamings. This is defined mutually with a law which says that it is compatible with the weakening substitution. The definitions are by induction on the renaming. Weakening basically adds one to all De Bruijn indices in the list of variables.

$$\begin{aligned}
&\mathsf{wkV} : && \mathsf{Vars}\,\Omega\,\Psi && \to \mathsf{Vars}\,(\Omega, A)\,\Psi \\
&\ulcorner\mathsf{wkV}\urcorner : && \{\beta : \mathsf{Vars}\,\Omega\,\Psi\} && \to \ulcorner\beta\urcorner \circ \pi_1\mathsf{id} \equiv \ulcorner\mathsf{wkV}\,\beta\urcorner \\
&\mathsf{wkV} && \epsilon && := \epsilon \\
&\mathsf{wkV} && (\beta, x) && := \mathsf{wkV}\,\beta,_{\ulcorner\mathsf{wkV}\urcorner *}\mathsf{vs}\,x
\end{aligned}$$

Now the identity renaming can be given mutually with its compatibility law. In the case of extended contexts, we use weakening of the identity for the first part of the context and then we add the zero De Bruijn index.

$$\begin{aligned}
&\mathsf{id} &&: \mathsf{Vars}\,\Psi\,\Psi \\
&\ulcorner\mathsf{id}\urcorner &&: \ulcorner\mathsf{id}\urcorner \equiv \mathsf{id} \\
&\mathsf{id}_{\cdot} &&:= \epsilon \\
&\mathsf{id}_{\Gamma,A} &&:= \mathsf{wkV}\,\mathsf{id},_{\ulcorner\mathsf{id}\urcorner *}\mathsf{vz}
\end{aligned}$$

We also define the weakening renaming which corresponds to the weakening substitution. Also, analogously with ↑ for substitutions, we define lifting of renamings.

$$\begin{aligned}
&\mathsf{wk} && : \mathsf{Vars}\,(\Psi, A)\,\Psi && := \mathsf{wkV}\,\mathsf{id} \\
&(\beta : \mathsf{Vars}\,\Omega\,\Psi) \uparrow A && : \mathsf{Vars}\,(\Omega, A[\ulcorner\beta\urcorner])\,(\Psi, A) && := \beta \circ \mathsf{wk}, \mathsf{vz}
\end{aligned}$$

We can now prove the following laws in the following order by induction on β or x completing the categorical structure.

$$\begin{aligned}
&\mathsf{wkV}\beta && : \mathsf{wkV}\,\beta \circ (\gamma, x) \equiv \beta \circ \gamma \\
&\mathsf{id}\circ && : \mathsf{id} \circ \beta \equiv \beta \\
&\mathsf{vs}[] && : \mathsf{vs}\,(x[\beta]) \equiv x[\mathsf{wkV}\,\beta] \\
&[\mathsf{id}] && : x[\mathsf{id}] \equiv x \\
&\circ\mathsf{id} && : \beta \circ \mathsf{id} \equiv \beta \\
&[][] && : x[\gamma][\beta] \equiv x[\gamma \circ \beta] \\
&\circ\circ && : (\iota \circ \gamma) \circ \beta \equiv \iota \circ (\gamma \circ \beta)
\end{aligned}$$

Note that what we are doing here is replicating all the laws of the syntax for a subset of the syntax. This subset is a full subcategory of the category of contexts and substitutions. The morphisms in this category (Vars) are given by a simple inductive type (without equality constructors).

5.7 The presheaf logical predicate interpretation

In this section, we define the proof-relevant presheaf logical predicate interpretation of the type theory given in chapter 3. The difference from the $-^{\mathsf{P}}$ operation defined in chapter 4 is that the target of this interpretation is the metatheory (not the object theory) and we define a Kripke version of the logical predicate: we define it over the base category REN.

5.7.1 Motives

First we define the Yoneda embedding of the syntax denoted TM. It has the following definitions for contexts, types, substitutions and terms.

$$\begin{array}{llll}
\Delta : \mathsf{Con} & \mathsf{TM}_\Delta : \mathsf{PSh}\,\mathsf{REN} & \mathsf{TM}_\Delta\,\Psi \;:= \mathsf{Tms}\,\Psi\,\Delta & \mathsf{TM}_\Delta\,\beta\,\rho := \rho \circ \ulcorner\beta\urcorner \\
A : \mathsf{Ty}\,\Gamma & \mathsf{TM}_A : \mathsf{FamPSh}\,\mathsf{TM}_\Gamma & \mathsf{TM}_{A\,\Psi}\,\rho := \mathsf{Tm}\,\Psi\,A[\rho] & \mathsf{TM}_A\,\beta\,t \;:= t[\ulcorner\beta\urcorner] \\
\sigma : \mathsf{Tms}\,\Gamma\,\Delta & \mathsf{TM}_\sigma : \mathsf{TM}_\Gamma \dot\to \mathsf{TM}_\Delta & \mathsf{TM}_{\sigma\,\Psi}\,\rho := \sigma \circ \rho & \\
t : \mathsf{Tm}\,\Gamma\,A & \mathsf{TM}_t \;: \mathsf{TM}_\Gamma \xrightarrow{\mathsf{s}} \mathsf{TM}_A & \mathsf{TM}_{t\,\Psi}\,\rho \;:= t[\rho] &
\end{array}$$

TM_Δ is a presheaf over REN. The functor laws hold as $\rho \circ \ulcorner\mathsf{id}\urcorner \equiv \rho$ and $(\rho \circ \ulcorner\beta\urcorner) \circ \ulcorner\gamma\urcorner \equiv \rho \circ \ulcorner\beta \circ \gamma\urcorner$. TM_A is a family of presheaves over TM_Γ, and similarly we have $t[\ulcorner\mathsf{id}\urcorner] \equiv t$ and $t[\ulcorner\beta\urcorner][\ulcorner\gamma\urcorner] \equiv t[\ulcorner\beta \circ \gamma\urcorner]$. TM_σ is a natural transformation, naturality is given by associativity: $(\sigma \circ \rho) \circ \ulcorner\beta\urcorner \equiv \sigma \circ (\rho \circ \ulcorner\beta\urcorner)$. TM_t is a section, it's naturality law can be verified as $t[\rho][\ulcorner\beta\urcorner] \equiv t[\rho \circ \ulcorner\beta\urcorner]$.

Intuitively TM embeds a context into the sets of substitutions into that context, a type into the sets of terms into that context, and substitutions and terms are embedded into maps between the corresponding sets.

TM is not the presheaf interpretation (section 5.4), it can be seen as a (weak) morphism in the category of models of type theory from the syntax to the presheaf model which is different from the presheaf interpretation (we don't give these notions a precise meaning in this thesis).

it is just the syntax in a different structure so that it matches the motives of the presheaf model.

The motives for the presheaf logical predicate interpretation are given as families over the Yoneda embedding TM. In contrast with the previous chapter, in this chapter we use a recursive notation for defining the motives and methods.

$$\begin{array}{ll}
\Delta : \mathsf{Con} & \mathsf{P}_\Delta : \mathsf{FamPSh}\,\mathsf{TM}_\Delta \\
A : \mathsf{Ty}\,\Gamma & \mathsf{P}_A : \mathsf{FamPSh}\,\big(\Sigma\,(\Sigma\,\mathsf{TM}_\Gamma\,\mathsf{TM}_A)\,\mathsf{P}_\Gamma[\mathsf{wk}]\big) \\
\sigma \;: \mathsf{Tms}\,\Gamma\,\Delta & \mathsf{P}_\sigma \;: \Sigma\,\mathsf{TM}_\Gamma\,\mathsf{P}_\Gamma \xrightarrow{\mathsf{s}} \mathsf{P}_\Delta[\mathsf{TM}_\sigma][\mathsf{wk}] \\
t \;: \mathsf{Tm}\,\Gamma\,A & \mathsf{P}_t \;: \Sigma\,\mathsf{TM}_\Gamma\,\mathsf{P}_\Gamma \xrightarrow{\mathsf{s}} \mathsf{P}_A[\mathsf{TM}_t \uparrow \mathsf{P}_\Gamma]
\end{array}$$

For reference, the "arguments for the eliminator" notation for the motives looks as follows.

$$
\begin{aligned}
&\mathsf{Con}^{\mathsf{M}}\,\Delta && := \mathsf{FamPSh}\,\mathsf{TM}_{\Delta}\\
&\mathsf{Ty}^{\mathsf{M}}\,\Gamma^{M}\,A && := \mathsf{FamPSh}\,\big(\Sigma\,(\Sigma\,\mathsf{TM}_{\Gamma}\,\mathsf{TM}_{A})\,\Gamma^{M}[\mathsf{wk}]\big)\\
&\mathsf{Tms}^{\mathsf{M}}\,\Gamma^{M}\,\Delta^{M}\,\sigma && := \Sigma\,\mathsf{TM}_{\Gamma}\,\Gamma^{M} \xrightarrow{\mathsf{s}} \Delta^{M}[\mathsf{TM}_{\sigma}][\mathsf{wk}]\\
&\mathsf{Tm}^{\mathsf{M}}\,\Gamma^{M}\,A^{M}\,t && := \Sigma\,\mathsf{TM}_{\Gamma}\,\Gamma^{M} \xrightarrow{\mathsf{s}} A^{M}[\mathsf{TM}_{t} \uparrow \Gamma^{M}]
\end{aligned}
$$

We unfold these definitions a bit below.

In the logical predicate interpretation, a context Δ is mapped to a family of presheaves over TM_{Δ}. That is, for every substitution $\rho : \mathsf{TM}_{\Delta}\,\Psi$ we have a type $\mathsf{P}_{\Delta\,\Psi}\,\rho$ expressing that the logical predicate holds for ρ. Moreover, we have the renaming $\mathsf{P}_{\Gamma}\,\beta : \mathsf{P}_{\Gamma}\,\rho \to \mathsf{P}_{\Gamma}\,(\mathsf{TM}_{\Gamma}\,\beta\,\rho)$ for a $\beta : \mathsf{REN}(\Omega, \Psi)$.

P_{A} is the logical predicate at a type A. It depends on a substitution (for which the predicate needs to hold as well) and a term. $\mathsf{P}_{A\,\Psi}\,(\rho, s, \alpha)$ expresses that the logical predicate holds for term $s : \mathsf{Tm}\,\Psi\,A[\rho]$.

$$
\frac{A : \mathsf{Ty}\,\Gamma \qquad \Psi : |\mathsf{REN}| \qquad \rho : \mathsf{TM}_{\Gamma}\,\Psi \qquad s : \mathsf{TM}_{A}\,\rho \qquad \alpha : \mathsf{P}_{\Gamma\,\Psi}\,\rho}{\mathsf{P}_{A\,\Psi}\,(\rho, s, \alpha) : \mathsf{Set}}
$$

It is also stable under renamings.

$$
\mathsf{P}_{A}\,\beta : \mathsf{P}_{A}\,(\rho, s, \alpha) \to \mathsf{P}_{A}\,(\mathsf{TM}_{\Gamma}\,\beta\,\rho, \mathsf{TM}_{A}\,\beta\,s, \mathsf{P}_{\Gamma}\,\beta\,\alpha)
$$

A substitution σ is mapped to P_{σ} which expresses the fundamental theorem of the logical predicate at σ: for any other substitution ρ for which the predicate holds, we can compose it with σ and the predicate will hold for the composition.

$$
\frac{\sigma : \mathsf{Tms}\,\Gamma\,\Delta \qquad \Psi : |\mathsf{REN}| \qquad \rho : \mathsf{TM}_{\Gamma}\,\Psi \qquad \alpha : \mathsf{P}_{\Gamma\,\Psi}\,\rho}{\mathsf{P}_{\sigma\,\Psi}\,(\rho, \alpha) : \mathsf{P}_{\Delta\,\Psi}\,(\sigma \circ \rho)}
$$

The fundamental theorem is also natural.

$$
\mathsf{P}_{\Delta}\,\beta\,(\mathsf{P}_{\sigma}\,(\rho, \alpha)) \equiv \mathsf{P}_{\sigma}\,(\mathsf{TM}_{\Gamma}\,\beta\,\rho, \mathsf{P}_{\Gamma}\,\beta\,\alpha)
$$

A term t is mapped to the fundamental theorem at the term: given a substitution ρ for which the predicate holds, it also holds for $t[\rho]$ in a natural way.

$$\frac{t : \mathsf{Tm}\,\Gamma\,A \qquad \Psi : |\mathsf{REN}^{\mathsf{op}}| \qquad \rho : \mathsf{TM}_\Gamma\,\Psi \qquad \alpha : \mathsf{P}_{\Gamma\Psi}\,\rho}{\mathsf{P}_{t\Psi}\,(\rho,\alpha) : \mathsf{P}_{A\Psi}\,(\rho, t[\rho], \alpha)}$$

$$\mathsf{P}_A\,\beta\,\big(\mathsf{P}_t\,(\rho,\alpha)\big) \equiv \mathsf{P}_t\,(\mathsf{TM}_\Gamma\,\beta\,\rho, \mathsf{P}_\Gamma\,\beta\,\alpha)$$

5.7.2 Methods for the substitution calculus

The logical predicate trivially holds at the empty context and it holds at an extended context for ρ if it holds at the smaller context for $\pi_1\,\rho$ and if it holds at the type which extends the context for $\pi_2\,\rho$. The second part obviously depends on the first. The action on morphisms for context extension is pointwise.

$$\begin{aligned}
&\mathsf{P}_{\cdot\Psi}\,(\rho : \mathsf{TM}_{\cdot}\,\Psi) && := \top\\
&\mathsf{P}_{\cdot}\,\beta\,\_ && := \mathsf{tt}\\
&\mathsf{P}_{\Delta,A\Psi}\,(\rho : \mathsf{TM}_{\Delta,A}\,\Psi) && := \Sigma(\alpha : \mathsf{P}_{\Delta\Psi}\,(\pi_1\,\rho)).\mathsf{P}_{A\Psi}\,(\pi_1\,\rho, \pi_2\,\rho, \alpha)\\
&\mathsf{P}_{\Delta,A}\,(\beta : \mathsf{REN}(\Omega,\Psi))\,(\alpha, a) && := (\mathsf{P}_\Delta\,\beta\,\alpha, \mathsf{P}_A\,\beta\,a)
\end{aligned}$$

The functor laws hold by

$$\mathsf{P}_{\Delta,A}\,\mathsf{id}\,(\alpha,a) = (\mathsf{P}_\Delta\,\mathsf{id}\,\alpha, \mathsf{P}_A\,\mathsf{id}\,a) \equiv (\alpha, a)$$

and

$$\begin{aligned}
&\mathsf{P}_{\Delta,A}\,\gamma\,\big(\mathsf{P}_{\Delta,A}\,\beta\,(\alpha,a)\big) = \big(\mathsf{P}_\Delta\,\gamma\,(\mathsf{P}_\Delta\,\beta\,\alpha), \mathsf{P}_\Delta\,\gamma\,(\mathsf{P}_A\,\beta\,a)\big)\\
&\equiv (\mathsf{P}_\Delta\,(\beta\circ\gamma)\,\alpha, \mathsf{P}_A\,(\beta\circ\gamma)\,a) = \mathsf{P}_{\Delta,A}\,(\beta\circ\gamma)\,(\alpha,a).
\end{aligned}$$

The logical predicate at a substituted type is the logical predicate at the type and we need to use the fundamental theorem at the substitution to lift the witness of the predicate for the substitution. Renaming a substituted type is the same as renaming in the original type (hence the functor laws hold immediately by the inductive hypothesis). This is well-typed because of naturality of TM_σ and P_σ as shown below.

$$\mathsf{P}_{A[\sigma]}\,(\rho, s, \alpha) := \mathsf{P}_A\,\big(\mathsf{TM}_\sigma\,\rho, s, \mathsf{P}_\sigma\,(\rho,\alpha)\big)$$

$$\mathsf{P}_{A[\sigma]}\,\beta\,a \qquad := \mathsf{P}_A\,\beta\,a : \underbrace{\mathsf{P}_A\left(\mathsf{TM}_\Gamma\,\beta\,(\mathsf{TM}_\sigma\,\rho), \mathsf{TM}_A\,\beta\,s, \mathsf{P}_\Gamma\,\beta\,(\mathsf{P}_\sigma\,(\rho,\alpha))\right)}_{\equiv \mathsf{P}_A\left(\mathsf{TM}_\sigma\,(\mathsf{TM}_\Theta\,\beta\,\rho), \mathsf{TM}_A\,\beta\,s, \mathsf{P}_\sigma\,(\mathsf{TM}_\Theta\,\beta\,\rho, \mathsf{P}_\Theta\,\beta\,\alpha)\right)}$$

The methods for the substitution constructors map the object theoretic constructs to their metatheoretic counterparts: identity becomes identity, composition becomes composition, the empty substitution becomes the element of the unit type, comprehension becomes pairing, first projection becomes first projection.

$$\begin{aligned}
\mathsf{P}_{\mathsf{id}}\,(\rho,\alpha) &:= \alpha \\
\mathsf{P}_{\sigma\circ\nu}\,(\rho,\alpha) &:= \mathsf{P}_\sigma\left(\mathsf{TM}_\nu\,\rho, \mathsf{P}_\nu\,(\rho,\alpha)\right) \\
\mathsf{P}_\epsilon\,(\rho,\alpha) &:= \mathsf{tt} \\
\mathsf{P}_{\sigma,t}\,(\rho,\alpha) &:= \mathsf{P}_\sigma\,(\rho,\alpha), \mathsf{P}_t\,(\rho,\alpha) \\
\mathsf{P}_{\pi_1\,\sigma}\,(\rho,\alpha) &:= \mathsf{proj}_1\left(\mathsf{P}_\sigma\,(\rho,\alpha)\right)
\end{aligned}$$

We check naturality for each case (see the description after the motives for the general naturality rule). The usages of $\equiv$ are using the induction hypotheses, the usages of $=$ are just definitional equalities. Note that in one case we used the naturality of TM_ν.

$$\begin{aligned}
&\mathsf{P}_\Gamma\,\beta\left(\mathsf{P}_{\mathsf{id}}\,(\rho,\alpha)\right) = \mathsf{P}_\Gamma\,\beta\,\alpha = \mathsf{P}_{\mathsf{id}}\,(\mathsf{TM}_\Gamma\,\beta\,\rho, \mathsf{P}_\Gamma\,\beta\,\alpha) \\
&\mathsf{P}_\Delta\,\beta\left(\mathsf{P}_{\sigma\circ\nu}\,(\rho,\alpha)\right) \\
=\,&\mathsf{P}_\Delta\,\beta\left(\mathsf{P}_\sigma\,(\mathsf{TM}_\nu\,\rho, \mathsf{P}_\nu\,(\rho,\alpha))\right) \\
\equiv\,&\mathsf{P}_\sigma\left(\mathsf{TM}_\Theta\,\beta\,(\mathsf{TM}_\nu\,\rho), \mathsf{P}_\Theta\,\beta\,(\mathsf{P}_\nu\,(\rho,\alpha))\right) \\
\equiv\,&\mathsf{P}_\sigma\left(\mathsf{TM}_\nu\,(\mathsf{TM}_\Gamma\,\beta\,\rho), (\mathsf{P}_\nu\,(\mathsf{TM}_\Gamma\,\beta\,\rho, \mathsf{P}_\Gamma\,\beta\,\alpha))\right) \\
=\,&\mathsf{P}_{\sigma\circ\nu}\,(\mathsf{TM}_\Gamma\,\beta\,\rho, \mathsf{P}_\Gamma\,\beta\,\alpha) \\
&\mathsf{P}_{\cdot}\,\beta\left(\mathsf{P}_\epsilon\,(\rho,\alpha)\right) = \mathsf{tt} = \mathsf{P}_\epsilon\,(\mathsf{TM}_\Gamma\,\beta\,\rho, \mathsf{P}_\Gamma\,\beta\,\alpha) \\
&\mathsf{P}_{\Delta,A}\,\beta\left(\mathsf{P}_{\sigma,t}\,(\rho,\alpha)\right) \\
=\,&\mathsf{P}_\Delta\,\beta\left(\mathsf{P}_\sigma\,(\rho,\alpha)\right), \mathsf{P}_A\,\beta\left(\mathsf{P}_t\,(\rho,\alpha)\right) \\
\equiv\,&\mathsf{P}_\sigma\,(\mathsf{TM}_\Gamma\,\beta\,\rho, \mathsf{P}_\Gamma\,\beta\,\alpha), \mathsf{P}_t\,(\mathsf{TM}_\Gamma\,\beta\,\rho, \mathsf{P}_\Gamma\,\beta\,\alpha) \\
=\,&\mathsf{P}_{\sigma,t}\,(\mathsf{TM}_\Gamma\,\beta\,\rho, \mathsf{P}_\Gamma\,\beta\,\alpha) \\
&\mathsf{P}_\Delta\,\beta\left(\mathsf{P}_{\pi_1\,\sigma}\,(\rho,\alpha)\right) \\
=\,&\mathsf{P}_\Delta\,\beta\left(\mathsf{proj}_1\,(\mathsf{P}_\sigma\,(\rho,\alpha))\right)
\end{aligned}$$

$$
\begin{aligned}
&= \mathsf{proj}_1 \left(\mathsf{P}_{\Delta,A}\, \beta \,(\mathsf{P}_\sigma\,(\rho,\alpha))\right)\\
&\equiv \mathsf{proj}_1 \left(\mathsf{P}_\sigma\,(\mathsf{TM}_\Gamma\, \beta\, \rho, \mathsf{P}_\Gamma\, \beta\, \alpha)\right)\\
&= \mathsf{P}_{\pi_1\,\sigma}\,(\mathsf{TM}_\Gamma\, \beta\, \rho, \mathsf{P}_\Gamma\, \beta\, \alpha)
\end{aligned}
$$

The fundamental theorem for substituted terms and the second projection are again just composition and second projection.

$$
\begin{aligned}
\mathsf{P}_{t[\sigma]}\,(\rho,\alpha) &:= \mathsf{P}_t\,(\mathsf{TM}_\sigma\,\rho, \mathsf{P}_\sigma\,(\rho,\alpha))\\
\mathsf{P}_{\pi_2\,\sigma}\,(\rho,\alpha) &:= \mathsf{proj}_2 \left(\mathsf{P}_\sigma\,(\rho,\alpha)\right)
\end{aligned}
$$

We check the naturality laws.

$$
\begin{aligned}
&\mathsf{P}_{A[\sigma]}\, \beta\, (\mathsf{P}_{t[\sigma]}\,(\rho,\alpha))\\
&= \mathsf{P}_A\, \beta \left(\mathsf{P}_t\,(\mathsf{TM}_\sigma\,\rho, \mathsf{P}_\sigma\,(\rho,\alpha))\right)\\
&\equiv \mathsf{P}_t \left(\mathsf{TM}_\Theta\, \beta\,(\mathsf{TM}_\sigma\,\rho), \mathsf{P}_\Theta\, \beta\,(\mathsf{P}_\sigma\,(\rho,\alpha))\right)\\
&\equiv \mathsf{P}_t \left(\mathsf{TM}_\sigma\,(\mathsf{TM}_\Gamma\, \beta\, \rho), \mathsf{P}_\sigma\,(\mathsf{TM}_\Gamma\, \beta\, \rho, \mathsf{P}_\Gamma\, \beta\, \alpha)\right)\\
&= \mathsf{P}_{t[\sigma]}\,(\mathsf{TM}_\Gamma\, \beta\, \rho, \mathsf{P}_\Gamma\, \beta\, \alpha)
\end{aligned}
$$

$$
\begin{aligned}
&\mathsf{P}_A\, \beta \left(\mathsf{P}_{\pi_2\,\sigma}\,(\rho,\alpha)\right)\\
&=\mathsf{P}_A\, \beta \left(\mathsf{proj}_2\,(\mathsf{P}_\sigma\,(\rho,\alpha))\right)\\
&=\mathsf{proj}_2 \left(\mathsf{P}_{\Delta,A}\, \beta\,(\mathsf{P}_\sigma\,(\rho,\alpha))\right)\\
&\equiv\mathsf{proj}_2 \left(\mathsf{P}_\sigma\,(\mathsf{TM}_\Gamma\, \beta\, \rho, \mathsf{P}_\Gamma\, \beta\, \alpha)\right)\\
&=\mathsf{P}_{\pi_2\,\sigma}\,(\mathsf{TM}_\Gamma\, \beta\, \rho, \mathsf{P}_\Gamma\, \beta\, \alpha)
\end{aligned}
$$

We finished defining the methods for point constructors of the substitution calculus, now we need to check the equality methods.

Note that types are interpreted as families of presheaves. Two families of presheaves are equal if their action on objects and morphisms are equal, because the equalities will be equal by K. Hence we only check equality for the actions on objects and morphisms.

The object part of the law [][] is validated by associativity of composition of substitutions.

$$
\begin{aligned}
&\mathsf{P}_{A[\sigma][\nu]}\,(\rho, s, \alpha)\\
=&\,\mathsf{P}_{A[\sigma]}\,\big(\mathsf{TM}_\nu\,\rho, s, \mathsf{P}_\nu\,(\rho,\alpha)\big)\\
=&\,\mathsf{P}_{A}\,\big(\mathsf{TM}_\sigma\,(\mathsf{TM}_\nu\,\rho), s, \mathsf{P}_\sigma\,(\mathsf{TM}_\nu\,\rho, \mathsf{P}_\nu\,(\rho,\alpha))\big)\\
&\hspace{15em}(\circ\circ)\\
\equiv&\,\mathsf{P}_{A}\,\big(\mathsf{TM}_{\sigma\circ\nu}\,\rho, s, \mathsf{P}_\sigma\,(\mathsf{TM}_\nu\,\rho, \mathsf{P}_\nu\,(\rho,\alpha))\big)\\
=&\,\mathsf{P}_{A}\,\big(\mathsf{TM}_{\sigma\circ\nu}\,\rho, s, \mathsf{P}_{\sigma\circ\nu}\,(\rho,\alpha)\big)\\
=&\,\mathsf{P}_{A[\sigma\circ\nu]}\,(\rho, s, \alpha)
\end{aligned}
$$

The object part of [id] is validated by the identity law of substitutions.

$$
\begin{aligned}
&\mathsf{P}_{A[\mathsf{id}]}\,(\rho, s, \alpha)\\
=&\,\mathsf{P}_{A}\,\big(\mathsf{TM}_{\mathsf{id}}\,\rho, s, \mathsf{P}_{\mathsf{id}}\,(\rho,\alpha)\big)\\
=&\,\mathsf{P}_{A}\,\big(\mathsf{TM}_{\mathsf{id}}\,\rho, s, \alpha\big)\\
&\hspace{10em}(\mathsf{id}\circ)\\
\equiv&\,\mathsf{P}_{A}\,(\rho, s, \alpha)
\end{aligned}
$$

The morphism parts are immediate.

$$
\begin{aligned}
&\mathsf{P}_{A[\sigma][\nu]}\,\beta\,a = \mathsf{P}_{A}\,\beta\,a = \mathsf{P}_{A[\sigma\circ\nu]}\,\beta\,a\\
&\mathsf{P}_{A[\mathsf{id}]}\,\beta\,a = \mathsf{P}_{A}\,\beta\,a
\end{aligned}
$$

Note that substitutions are interpreted as sections. Two sections are equal if their function parts are equal, as the naturality conditions will be equal by K. We validate the laws for Tms below. Apart from the two steps where we use $\circ\circ$ and $\mathsf{id}\circ$ everything is definitional (as we have definitional η for $\top$ and Σ in our metatheory).

$$
\begin{aligned}
\circ\circ^{\mathsf{M}}\ :\ &\mathsf{P}_{(\sigma\circ\nu)\circ\delta}\,(\rho,\alpha)\\
=&\,\mathsf{P}_{\sigma\circ\nu}\,(\mathsf{TM}_\delta\,\rho, \mathsf{P}_\delta\,(\rho,\alpha))\\
=&\,\mathsf{P}_{\sigma}\,\big(\mathsf{TM}_\nu\,(\mathsf{TM}_\delta\,\rho), \mathsf{P}_\nu\,(\mathsf{TM}_\delta\,\rho, \mathsf{P}_\delta\,(\rho,\alpha))\big)\\
&\hspace{15em}(\circ\circ)\\
\equiv&\,\mathsf{P}_{\sigma}\,\big(\mathsf{TM}_{\nu\circ\delta}\,\rho, \mathsf{P}_\nu\,(\mathsf{TM}_\delta\,\rho, \mathsf{P}_\delta\,(\rho,\alpha))\big)
\end{aligned}
$$

$$
\begin{aligned}
& && = \mathsf{P}_{\sigma}\left(\mathsf{TM}_{\nu\circ\delta}\,\rho, \mathsf{P}_{\nu\circ\delta}\,(\rho,\alpha)\right) \\
& && = \mathsf{P}_{\sigma\circ(\nu\circ\delta)}\,(\rho,\alpha) \\
& \mathsf{id}\circ^{\mathsf{M}} && : \ \mathsf{P}_{\mathsf{id}\circ\sigma}\,(\rho,\alpha) \\
& && = \mathsf{P}_{\mathsf{id}}\left(\mathsf{TM}_{\sigma}\,\rho, \mathsf{P}_{\sigma}\,(\rho,\alpha)\right) \\
& && = \mathsf{P}_{\sigma}\,(\rho,\alpha) \\
& \circ\mathsf{id}^{\mathsf{M}} && : \ \mathsf{P}_{\sigma\circ\mathsf{id}}\,(\rho,\alpha) \\
& && = \mathsf{P}_{\sigma}\left(\mathsf{TM}_{\mathsf{id}}\,\rho, \mathsf{P}_{\mathsf{id}}\,(\rho,\alpha)\right) \\
& && = \mathsf{P}_{\sigma}\left(\mathsf{TM}_{\mathsf{id}}\,\rho, \alpha)\right) \\
& && \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad (\mathsf{id}\circ) \\
& && \equiv \mathsf{P}_{\sigma}\,(\rho,\alpha)\big) \\
& \epsilon\eta^{\mathsf{M}} && \{\sigma : \mathsf{Tms}\,\Gamma\,\cdot\} : \mathsf{P}_{\sigma}\,(\rho,\alpha) = \mathsf{tt} : \underbrace{\mathsf{P}.\,\rho}_{=\top} \qquad \text{(metatheoretic } \eta \text{ for } \top) \\
& \pi_1\beta^{\mathsf{M}} && : \ \mathsf{P}_{\pi_1\,(\sigma,t)}\,(\rho,\alpha) \\
& && = \mathsf{proj}_1\left(\mathsf{P}_{\sigma,t}\,(\rho,\alpha)\right) \\
& && = \mathsf{proj}_1\left(\mathsf{P}_{\sigma}\,(\rho,\alpha), \mathsf{P}_{t}\,(\rho,\alpha)\right) \\
& && = \mathsf{P}_{\sigma}\,(\rho,\alpha) \\
& \pi\eta^{\mathsf{M}} && : \ \mathsf{P}_{\pi_1\,\sigma,\pi_2\,\sigma}\,(\rho,\alpha) \\
& && = \mathsf{P}_{\pi_1\,\sigma}\,(\rho,\alpha), \mathsf{P}_{\pi_2\,\sigma}\,(\rho,\alpha) \\
& && = \mathsf{proj}_1\left(\mathsf{P}_{\sigma}\,(\rho,\alpha)\right), \mathsf{proj}_2\left(\mathsf{P}_{\sigma}\,(\rho,\alpha)\right) \\
& && \qquad\qquad\qquad\qquad\qquad\qquad \text{(metatheoretic } \eta \text{ for } \Sigma) \\
& && = \mathsf{P}_{\sigma}\,(\rho,\alpha) \\
& ,\circ^{\mathsf{M}} && : \ \mathsf{P}_{(\nu,t)\circ\sigma}\,(\rho,\alpha) \\
& && = \mathsf{P}_{\nu,t}\left(\mathsf{TM}_{\sigma}\,\rho, \mathsf{P}_{\sigma}\,(\rho,\alpha)\right) \\
& && = \mathsf{P}_{\nu}\left(\mathsf{TM}_{\sigma}\,\rho, \mathsf{P}_{\sigma}\,(\rho,\alpha)\right), \mathsf{P}_{t}\left(\mathsf{TM}_{\sigma}\,\rho, \mathsf{P}_{\sigma}\,(\rho,\alpha)\right) \\
& && = \mathsf{P}_{\nu\circ\sigma}\,(\rho,\alpha), \mathsf{P}_{t[\sigma]}\,(\rho,\alpha) \\
& && = \mathsf{P}_{\nu\circ\sigma,t[\sigma]}\,(\rho,\alpha)
\end{aligned}
$$

Finally we verify the single equality rule for terms, here again, it is enough to verify that the function parts are equal and the naturality conditions will be equal by K.

$$\begin{aligned}\pi_2\beta^{\mathsf{M}} :\ & \mathsf{P}_{\pi_2\,(\sigma,t)}\,(\rho,\alpha)\\ =&\mathsf{proj}_2\,\big(\mathsf{P}_{\sigma,t}\,(\rho,\alpha)\big)\\ =&\mathsf{proj}_2\,\big(\mathsf{P}_{\sigma}\,(\rho,\alpha),\mathsf{P}_t\,(\rho,\alpha)\big)\\ =&\mathsf{P}_t\,(\rho,\alpha)\end{aligned}$$

We state the following equalities which will be useful in the next sections, their proofs are trivial.

$$\begin{aligned}\mathsf{P}_{\mathsf{wk}}\,(\rho,\alpha) &\equiv \mathsf{proj}_1\,\alpha\\ \mathsf{P}_{\mathsf{vz}}\,(\rho,\alpha) &\equiv \mathsf{proj}_2\,\alpha\\ \mathsf{P}_{\uparrow} : \mathsf{P}_{\sigma\uparrow A}\,(\rho,\alpha) &\equiv \mathsf{P}_{\sigma}\,(\pi_1\,\rho,\mathsf{proj}_1\,\alpha),\mathsf{proj}_2\,\alpha\\ \mathsf{P}_{\langle t\rangle}\,(\rho,\alpha) &\equiv \alpha,\mathsf{P}_t\,(\rho,\alpha)\end{aligned}$$

5.7.3 Methods for the base type and base family

We extend the Yoneda embedding TM with two special constructs, one for the base type U and one for the base family El. These express that the base type is closed, hence a presheaf is enough to define its Yoneda embedding, we don't need a family of presheaves.

$$\begin{array}{lll}\mathsf{TM}^{\mathsf{U}} : \mathsf{PSh}\,\mathsf{REN} & \mathsf{TM}^{\mathsf{U}}\,\Psi := \mathsf{Tm}\,\Psi\,\mathsf{U} & \mathsf{TM}^{\mathsf{U}}\,\beta\,\hat{A} := {}_{\mathsf{U}[]*}\hat{A}[\ulcorner\beta\urcorner]\\ \mathsf{TM}^{\mathsf{El}} : \mathsf{FamPSh}\,\mathsf{TM}^{\mathsf{U}} & \mathsf{TM}^{\mathsf{El}}{}_{\Psi}\,\hat{A} := \mathsf{Tm}\,\Psi\,(\mathsf{El}\,\hat{A}) & \mathsf{TM}^{\mathsf{El}}\,\beta\,a := {}_{\mathsf{El}[]*}a[\ulcorner\beta\urcorner]\end{array}$$

The functor laws follow from $[][]$ and $[\mathsf{id}]$.

We parameterise the logical predicate interpretation by the predicate at the base type U and base family El. These are denoted by $\bar{\mathsf{U}}$ and $\bar{\mathsf{El}}$ and have the following types.

$$\begin{aligned}\bar{\mathsf{U}} &: \mathsf{FamPSh}\,\mathsf{TM}^{\mathsf{U}}\\ \bar{\mathsf{El}} &: \mathsf{FamPSh}\,\big(\Sigma\,(\Sigma\,\mathsf{TM}^{\mathsf{U}}\,\mathsf{TM}^{\mathsf{El}})\,\bar{\mathsf{U}}[\mathsf{wk}]\big)\end{aligned}$$

The logical predicate at the base type and family says what we have given as parameters. Renaming also comes from these parameters.

$$\begin{aligned}
\mathsf{P}_{\mathsf{U}}\,(\rho, s, \alpha) &:= \bar{\mathsf{U}}\,({}_{\mathsf{U}[]*}s) & \mathsf{P}_{\mathsf{U}}\,\beta\,a &:= \bar{\mathsf{U}}\,\beta\,a \\
\mathsf{P}_{\mathsf{El}\,\hat{A}}\,(\rho, s, \alpha) &:= \bar{\mathsf{El}}\,\big(({}_{\mathsf{El}[]*}\mathsf{TM}_{\hat{A}}\,\rho), s, \mathsf{P}_{\hat{A}}\,(\rho, \alpha)\big) & \mathsf{P}_{\mathsf{El}\,\hat{A}}\,\beta\,a &:= \bar{\mathsf{El}}\,\beta\,a
\end{aligned}$$

The functor laws follow from the functor laws of the parameters.

We first verify the object parts of equality laws.

$$\begin{aligned}
\mathsf{U}[]^{\mathsf{M}} :\;\; & \mathsf{P}_{\mathsf{U}[\sigma]}\,(\rho, s, \alpha) \\
&= \mathsf{P}_{\mathsf{U}}\,\big(\mathsf{TM}_{\sigma}\,\rho, s, \mathsf{P}_{\sigma}\,(\rho, \alpha)\big) \\
&= \bar{\mathsf{U}}\,s \\
&= \mathsf{P}_{\mathsf{U}}\,(\rho, s, \alpha) \\
\mathsf{El}[]^{\mathsf{M}} :\;\; & \mathsf{P}_{(\mathsf{El}\,\hat{A})[\sigma]}\,(\rho, s, \alpha) \\
&= \mathsf{P}_{\mathsf{El}\,\hat{A}}\,\big(\mathsf{TM}_{\sigma}\,\rho, s, \mathsf{P}_{\sigma}\,(\rho, \alpha)\big) \\
&= \bar{\mathsf{El}}\,\big(\mathsf{TM}_{\hat{A}}\,(\mathsf{TM}_{\sigma}\,\rho), s, \mathsf{P}_{\hat{A}}\,(\mathsf{TM}_{\sigma}\,\rho, \mathsf{P}_{\sigma}\,(\rho, \alpha))\big) \\
& \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad (\text{[][] for terms}) \\
&\equiv \bar{\mathsf{El}}\,\big(\mathsf{TM}_{\hat{A}[\sigma]}\,\rho, s, \mathsf{P}_{\hat{A}}\,(\mathsf{TM}_{\sigma}\,\rho, \mathsf{P}_{\sigma}\,(\rho, \alpha))\big) \\
&= \bar{\mathsf{El}}\,\big(\mathsf{TM}_{\hat{A}[\sigma]}\,\rho, s, \mathsf{P}_{\hat{A}[\sigma]}\,(\rho, \alpha)\big) \\
&= \mathsf{P}_{\mathsf{El}\,(\hat{A}[\sigma])}\,(\rho, s, \alpha)
\end{aligned}$$

The morphism parts are given below.

$$\begin{aligned}
\mathsf{U}[]^{\mathsf{M}} &: \mathsf{P}_{\mathsf{U}[\sigma]}\,\beta\,a = \mathsf{P}_{\mathsf{U}}\,\beta\,a \\
\mathsf{El}[]^{\mathsf{M}} &: \mathsf{P}_{(\mathsf{El}\,\hat{A})[\sigma]}\,\beta\,a = \mathsf{P}_{\mathsf{El}\,\hat{A}}\,\beta\,a = \bar{\mathsf{El}}\,\beta\,a = \mathsf{P}_{\mathsf{El}\,(\hat{A}[\sigma])}\,\beta\,a
\end{aligned}$$

We have completed the methods for the base type and family by adding two parameters to the logical predicate interpretation: $\bar{\mathsf{U}}$ and $\bar{\mathsf{El}}$.

5.7.4 Methods for the function space

The logical predicate holds for a function s when we have that if the predicate holds for an argument u (at A, witnessed by v), so it holds for $s\$u$ at B. In addition, we have a Kripke style generalisation: this should be true for $\mathsf{TM}_{\Pi\,A\,B}\,\beta\,s$ for any morphism β in a natural way. Renaming a witness of the logical predicate at the function type is postcomposing the Kripke morphism by it.

$$
\begin{aligned}
&\mathsf{P}_{\Pi\,A\,B\,\Psi}\,\big((\rho : \mathsf{TM}_{\Gamma}\,\Psi), (s : \mathsf{TM}_{\Pi\,A\,B}\,\rho), (\alpha : \mathsf{P}_{\Gamma}\,\rho)\big)\\
&\qquad := \Sigma\Big(\mathsf{map} : \big(\beta : \mathsf{REN}(\Omega, \Psi)\big)\big(u : \mathsf{TM}_{A}\,(\mathsf{TM}_{\Gamma}\,\beta\,\rho)\big)\\
&\qquad\qquad\qquad\qquad \big(v : \mathsf{P}_{A\,\Omega}\,(\mathsf{TM}_{\Gamma}\,\beta\,\rho, u, \mathsf{P}_{\Gamma}\,\beta\,\alpha)\big)\\
&\qquad\qquad\qquad\qquad \to \mathsf{P}_{B\,\Omega}\,\big((\mathsf{TM}_{\Gamma}\,\beta\,\rho, u), (\mathsf{TM}_{\Pi\,A\,B}\,\beta\,s)\$u, (\mathsf{P}_{\Gamma}\,\beta\,\alpha, v)\big)\Big)\\
&\qquad\qquad\quad .\forall\beta, u, v, \gamma.\mathsf{P}_{B}\,\gamma\,(\mathsf{map}\,\beta\,u\,v) \equiv \mathsf{map}\,(\beta \circ \gamma)\,(\mathsf{TM}_{A}\,\gamma\,u)\,(\mathsf{P}_{A}\,\gamma\,v)\\
&\mathsf{P}_{\Pi\,A\,B}\,\beta'\,(\mathsf{map}, \mathsf{nat}) := \lambda\beta.\mathsf{map}\,(\beta' \circ \beta), \lambda\beta.\mathsf{nat}\,(\beta' \circ \beta)
\end{aligned}
$$

From the action on morphisms we calculate the following law which we will use later.

$$\mathsf{map}\circ : \mathsf{map}\,(\mathsf{P}_{\Pi\,A\,B}\,\beta'\,a)\,\beta = \mathsf{map}\,a\,(\beta' \circ \beta)$$

The functor laws follow from the categorical laws of REN.

$$
\begin{aligned}
&\mathsf{P}_{\Pi\,A\,B}\,\mathsf{id}\,(\mathsf{map}, \mathsf{nat})\\
=&\,\lambda\beta.\mathsf{map}\,(\mathsf{id} \circ \beta), \lambda\beta.\mathsf{nat}\,(\mathsf{id} \circ \beta)\\
\equiv&\,\lambda\beta.\mathsf{map}\,\beta, \lambda\beta.\mathsf{nat}\,\beta\\
=&\,(\mathsf{map}, \mathsf{nat})
\end{aligned}
$$

$$
\begin{aligned}
&\mathsf{P}_{\Pi\,A\,B}\,(\beta' \circ \gamma)\,(\mathsf{map}, \mathsf{nat})\\
=&\lambda\beta.\mathsf{map}\,\big((\beta' \circ \gamma) \circ \beta\big), \lambda\beta.\mathsf{nat}\,\big((\beta' \circ \gamma) \circ \beta\big)\\
\equiv&\lambda\beta.\mathsf{map}\,\big(\beta' \circ (\gamma \circ \beta)\big), \lambda\beta.\mathsf{nat}\,\big(\beta' \circ (\gamma \circ \beta)\big)\\
=&\mathsf{P}_{\Pi\,A\,B}\,\gamma\,\big(\lambda\beta.\mathsf{map}\,(\beta' \circ \beta), \lambda\beta.\mathsf{nat}\,(\beta' \circ \beta)\big)\\
=&\mathsf{P}_{\Pi\,A\,B}\,\gamma\,\big(\mathsf{P}_{\Pi\,A\,B}\,\beta'\,(\mathsf{map}, \mathsf{nat})\big)
\end{aligned}
$$

The object part of the interpretation of $\Pi[]$ can be verified as follows.

$$
\begin{aligned}
&\mathsf{P}_{(\Pi\,A\,B)[\sigma]}\,(\rho, s, \alpha)\\
=&\,\mathsf{P}_{\Pi\,A\,B}\,\big(\mathsf{TM}_{\sigma}\,\rho, s, \mathsf{P}_{\sigma}\,(\rho, \alpha)\big)\\
=&\,\Sigma\Big(\mathsf{map} : \big(\beta : \mathsf{REN}(\Omega, \Psi)\big)\big(u : \mathsf{TM}_{A}\,(\mathsf{TM}_{\Theta}\,\beta\,(\mathsf{TM}_{\sigma}\,\rho))\big)\\
&\qquad\qquad \big(v : \mathsf{P}_{A}\,(\mathsf{TM}_{\Theta}\,\beta\,(\mathsf{TM}_{\sigma}\,\rho), u, \mathsf{P}_{\Theta}\,\beta\,(\mathsf{P}_{\sigma}\,(\rho, \alpha)))\big)
\end{aligned}
$$

$$
\begin{aligned}
&\qquad\quad \to \mathsf{P}_B\,\big((\mathsf{TM}_\Theta\,\beta\,(\mathsf{TM}_\sigma\,\rho),u),(\mathsf{TM}_{\Pi\,A\,B}\,\beta\,s)\$u\\
&\qquad\qquad\quad ,(\mathsf{P}_\Theta\,\beta\,(\mathsf{P}_\sigma\,(\rho,\alpha)),v))\Big)\\
&\quad .\forall\beta,u,v,\gamma.\mathsf{P}_B\,\gamma\,(\mathsf{map}\,\beta\,u\,v)\equiv\mathsf{map}\,(\beta\circ\gamma)\,(\mathsf{TM}_A\,\gamma\,u)\,(\mathsf{P}_A\,\gamma\,v)\\
&\hfill(\text{naturality of }\mathsf{TM}_\sigma\text{ and }\mathsf{P}_\sigma)\\
\equiv&\,\Sigma\Big(\mathsf{map}:\big(\beta:\mathsf{REN}(\Omega,\Psi)\big)\big(u:\mathsf{TM}_A\,(\mathsf{TM}_\sigma\,(\mathsf{TM}_\Gamma\,\beta\,\rho))\big)\\
&\qquad\quad\big(v:\mathsf{P}_A\,(\mathsf{TM}_\sigma\,(\mathsf{TM}_\Gamma\,\beta\,\rho),u,\mathsf{P}_\sigma\,(\mathsf{TM}_\Gamma\,\beta\,\rho,\mathsf{P}_\Gamma\,\beta\,\alpha))\big)\\
&\qquad\quad \to \mathsf{P}_B\,\big((\mathsf{TM}_\sigma\,(\mathsf{TM}_\Gamma\,\beta\,\rho),u),(\mathsf{TM}_{\Pi\,A\,B}\,\beta\,s)\$u\\
&\qquad\qquad\quad ,(\mathsf{P}_\sigma\,(\mathsf{TM}_\Gamma\,\beta\,\rho,\mathsf{P}_\Gamma\,\beta\,\alpha),v))\Big)\\
&\quad .\forall\beta,u,v,\gamma.\mathsf{P}_B\,\gamma\,(\mathsf{map}\,\beta\,u\,v)\equiv\mathsf{map}\,(\beta\circ\gamma)\,(\mathsf{TM}_A\,\gamma\,u)\,(\mathsf{P}_A\,\gamma\,v)\\
&\hfill(\text{substitution calculus and }\mathsf{P}_\sigma)\\
\equiv&\,\Sigma\Big(\mathsf{map}:\big(\beta:\mathsf{REN}(\Omega,\Psi)\big)\big(u:\mathsf{TM}_{A[\sigma]}\,(\mathsf{TM}_\Gamma\,\beta\,\rho)\big)\\
&\qquad\quad\big(v:\mathsf{P}_A\,(\mathsf{TM}_\sigma\,(\mathsf{TM}_\Gamma\,\beta\,\rho),u,\mathsf{P}_\sigma\,(\mathsf{TM}_\Gamma\,\beta\,\rho,\mathsf{P}_\Gamma\,\beta\,\alpha))\big)\\
&\qquad\quad \to \mathsf{P}_B\,\big(\mathsf{TM}_{\sigma\uparrow}\,(\mathsf{TM}_\Gamma\,\beta\,\rho,u),(\mathsf{TM}_{\Pi\,A[\sigma]\,B[\sigma\uparrow]}\,\beta\,s)\$u\\
&\qquad\qquad\quad ,\mathsf{P}_{\sigma\uparrow}\,((\mathsf{TM}_\Gamma\,\beta\,\rho,u),(\mathsf{P}_\Gamma\,\beta\,\alpha,v)))\Big)\\
&\quad .\forall\beta,u,v,\gamma.\mathsf{P}_B\,\gamma\,(\mathsf{map}\,\beta\,u\,v)\equiv\mathsf{map}\,(\beta\circ\gamma)\,(\mathsf{TM}_{A[\sigma]}\,\gamma\,u)\,(\mathsf{P}_A\,\gamma\,v)\\
=&\,\Sigma\Big(\mathsf{map}:\big(\beta:\mathsf{REN}(\Omega,\Psi)\big)\big(u:\mathsf{TM}_{A[\sigma]}\,(\mathsf{TM}_\Gamma\,\beta\,\rho)\big)\\
&\qquad\quad\big(v:\mathsf{P}_{A[\sigma]}\,(\mathsf{TM}_\Gamma\,\beta\,\rho,u,\mathsf{P}_\Gamma\,\beta\,\alpha)\big)\\
&\qquad\quad \to \mathsf{P}_{B[\sigma\uparrow]}\,\big((\mathsf{TM}_\Gamma\,\beta\,\rho,u),(\mathsf{TM}_{\Pi\,A[\sigma]\,B[\sigma\uparrow]}\,\beta\,s)\$u,(\mathsf{P}_\Gamma\,\beta\,\alpha,v)\big)\Big)\\
&\quad .\forall\beta,u,v,\gamma.\mathsf{P}_{B[\sigma\uparrow]}\,\gamma\,(\mathsf{map}\,\beta\,u\,v)\equiv\mathsf{map}\,(\beta\circ\gamma)\,(\mathsf{TM}_{A[\sigma]}\,\gamma\,u)\,(\mathsf{P}_{A[\sigma]}\,\gamma\,v)\\
=&\,\mathsf{P}_{\Pi\,A[\sigma]\,B[\sigma\uparrow]}\,(\rho,s,\alpha)
\end{aligned}
$$

We verify the morphism part of Π[] below.

$$
\begin{aligned}
&\mathsf{P}_{(\Pi\,A\,B)[\sigma]}\,\beta'\,(\mathsf{map},\mathsf{nat})\\
=&\,\mathsf{P}_{\Pi\,A\,B}\,\beta'\,(\mathsf{map},\mathsf{nat})\\
=&\,\lambda\beta.\mathsf{map}\,(\beta'\circ\beta),\lambda\beta.\mathsf{nat}\,(\beta'\circ\beta)\\
=&\,\mathsf{P}_{\Pi\,A[\sigma]\,B[\sigma\uparrow]}\,\beta'\,(\mathsf{map},\mathsf{nat})
\end{aligned}
$$

Now we prove the fundamental theorem for lam and app (i.e. provide the corresponding methods). For lam, the map function is using the fundamental theorem for t which is in the context extended by the domain type $A : \mathsf{Ty}\,\Gamma$, so we need to supply an extended substitution and a witness of the predicate. Moreover, we need to rename the substitution ρ and the witness of the predicate α to account for the Kripke property. The naturality is given by the naturality of the term itself.

$$\begin{aligned}\mathsf{P}_{\mathsf{lam}\,t}\,(\rho,\alpha) := \Big(&\lambda\beta\,u\,v.\mathsf{P}_t\,\big((\mathsf{TM}_\Gamma\,\beta\,\rho,u),(\mathsf{P}_\Gamma\,\beta\,\alpha,v)\big)\\ &,\lambda\beta\,u\,v\,\gamma.\mathsf{natS}\,\mathsf{P}_t\,\big((\mathsf{TM}_\Gamma\,\beta\,\rho,u),(\mathsf{P}_\Gamma\,\beta\,\alpha,v)\big)\,\gamma\Big)\end{aligned}$$

Application uses the map part of the logical predicate and the identity renaming.

$$\mathsf{P}_{\mathsf{app}\,t}\,(\rho,\alpha) := \mathsf{map}\,\big(\mathsf{P}_t\,(\pi_1\,\rho,\mathsf{proj}_1\,\alpha)\big)\,\mathsf{id}\,(\pi_2\,\rho)\,(\mathsf{proj}_2\,\alpha)$$

We verify naturality of the interpretation of lam.

$$\begin{aligned}&\mathsf{P}_{\Pi\,A\,B}\,\beta'\,\big(\mathsf{P}_{\mathsf{lam}\,t}\,(\rho,\alpha)\big)\\ =\,&\mathsf{P}_{\Pi\,A\,B}\,\beta'\,\Big(\lambda\beta\,u\,v.\mathsf{P}_t\,\big((\mathsf{TM}_\Gamma\,\beta\,\rho,u),(\mathsf{P}_\Gamma\,\beta\,\alpha,v)\big)\\ &\qquad\qquad,\lambda\beta\,u\,v\,\gamma.\mathsf{natS}\,\mathsf{P}_t\,\big((\mathsf{TM}_\Gamma\,\beta\,\rho,u),(\mathsf{P}_\Gamma\,\beta\,\alpha,v)\big)\,\gamma\Big)\\ =\,&\ \lambda\beta\,u\,v.\mathsf{P}_t\,\big((\mathsf{TM}_\Gamma\,(\beta'\circ\beta)\,\rho,u),(\mathsf{P}_\Gamma\,(\beta'\circ\beta)\,\alpha,v)\big)\\ &\ ,\lambda\beta\,u\,v\,\gamma.\mathsf{natS}\,\mathsf{P}_t\,\big((\mathsf{TM}_\Gamma\,(\beta'\circ\beta)\,\rho,u),(\mathsf{P}_\Gamma\,(\beta'\circ\beta)\,\alpha,v)\big)\,\gamma\\ &\qquad\qquad\qquad\qquad\qquad\qquad(\text{functoriality of }\mathsf{TM}_\Gamma\text{ and }\mathsf{P}_\Gamma)\\ \equiv\,&\ \lambda\beta\,u\,v.\mathsf{P}_t\,\big((\mathsf{TM}_\Gamma\,\beta\,(\mathsf{TM}_\Gamma\,\beta'\,\rho),u),(\mathsf{P}_\Gamma\,\beta\,(\mathsf{P}_\Gamma\,\beta'\,\alpha),v)\big)\\ &\ ,\lambda\beta\,u\,v\,\gamma.\mathsf{natS}\,\mathsf{P}_t\,\big((\mathsf{TM}_\Gamma\,\beta\,(\mathsf{TM}_\Gamma\,\beta'\,\rho),u),(\mathsf{P}_\Gamma\,\beta\,(\mathsf{P}_\Gamma\,\beta'\,\alpha),v)\big)\,\gamma\\ =\,&\mathsf{P}_{\mathsf{lam}\,t}\,(\mathsf{TM}_\Gamma\,\beta'\,\rho,\mathsf{P}_\Gamma\,\beta'\,\alpha)\end{aligned}$$

Verifying the naturality of the interpretation of $\mathsf{app}\,t$ is more interesting: we need to use naturality of P_t and naturality of the mapping function in P_t at a given semantic element.

$$\begin{aligned}&\mathsf{P}_B\,\beta\,\big(\mathsf{P}_{\mathsf{app}\,t}\,(\rho,\alpha)\big)\\ =\,&\mathsf{P}_B\,\beta\,\big(\mathsf{map}\,\big(\mathsf{P}_t\,(\pi_1\,\rho,\mathsf{proj}_1\,\alpha)\big)\,\mathsf{id}\,(\pi_2\,\rho)\,(\mathsf{proj}_2\,\alpha)\big)\\ &\qquad\qquad\qquad\qquad\qquad\qquad(\text{naturality of }\mathsf{P}_t\,(\pi_1\,\rho,\mathsf{proj}_1\,\alpha))\\ \equiv\,&\mathsf{map}\,\big(\mathsf{P}_t\,(\pi_1\,\rho,\mathsf{proj}_1\,\alpha)\big)\,\beta\qquad\big(\mathsf{TM}_A\,\beta\,(\pi_2\,\rho)\big)\,\big(\mathsf{P}_A\,\beta\,(\mathsf{proj}_2\,\alpha)\big)\\ &\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad(\circ\mathsf{id})\end{aligned}$$

$$
\begin{aligned}
&\equiv \mathsf{map}\,\big(\mathsf{P}_t\,(\pi_1\,\rho, \mathsf{proj}_1\,\alpha)\big)\,(\beta \circ \mathsf{id})\,\big(\mathsf{TM}_A\,\beta\,(\pi_2\,\rho)\big)\,\big(\mathsf{P}_A\,\beta\,(\mathsf{proj}_2\,\alpha)\big) \\
&\hspace{20em} (\mathsf{map}\circ) \\
&\equiv \mathsf{map}\,\big(\mathsf{P}_{\Pi\,A\,B}\,\beta\,(\mathsf{P}_t\,(\pi_1\,\rho, \mathsf{proj}_1\,\alpha))\big)\,\mathsf{id}\,\big(\mathsf{TM}_A\,\beta\,(\pi_2\,\rho)\big)\,\big(\mathsf{P}_A\,\beta\,(\mathsf{proj}_2\,\alpha)\big) \\
&\hspace{20em} (\text{naturality of } \mathsf{P}_t) \\
&\equiv \mathsf{map}\,\Big(\mathsf{P}_t\,\big(\mathsf{TM}_\Gamma\,\beta\,(\pi_1\,\rho), \mathsf{P}_\Gamma\,\beta\,(\mathsf{proj}_1\,\alpha)\big)\Big)\,\mathsf{id}\,\big(\mathsf{TM}_A\,\beta\,(\pi_2\,\rho)\big) \\
&\qquad \big(\mathsf{P}_A\,\beta\,(\mathsf{proj}_2\,\alpha)\big) \\
&\hspace{20em} (\text{substitution calculus}) \\
&\equiv \mathsf{map}\,\Big(\mathsf{P}_t\,\big(\pi_1\,(\mathsf{TM}_{\Gamma,A}\,\beta\,\rho), \mathsf{P}_\Gamma\,\beta\,(\mathsf{proj}_1\,\alpha)\big)\Big)\,\mathsf{id}\,\big(\pi_2\,(\mathsf{TM}_{\Gamma,A}\,\beta\,\rho)\big) \\
&\qquad \big(\mathsf{P}_A\,\beta\,(\mathsf{proj}_2\,\alpha)\big) \\
&= \mathsf{map}\,\Big(\mathsf{P}_t\,\big(\pi_1\,(\mathsf{TM}_{\Gamma,A}\,\beta\,\rho), \mathsf{proj}_1\,(\mathsf{P}_{\Gamma,A}\,\beta\,\alpha)\big)\Big)\,\mathsf{id}\,\big(\pi_2\,(\mathsf{TM}_{\Gamma,A}\,\beta\,\rho)\big) \\
&\qquad \big(\mathsf{proj}_2\,(\mathsf{P}_{\Gamma,A}\,\beta\,\alpha)\big) \\
&= \mathsf{P}_{\mathsf{app}\,t}\,(\mathsf{TM}_{\Gamma,A}\,\beta\,\rho, \mathsf{P}_{\Gamma,A}\,\beta\,\alpha)
\end{aligned}
$$

Now we need to check the interpretations of the three equalities $\Pi\beta$, $\Pi\eta$ and $\mathsf{lam}[]$.

$$
\begin{aligned}
&\mathsf{P}_{\mathsf{app}\,(\mathsf{lam}\,t)}\,(\rho, \alpha) \\
&= \mathsf{map}\,\big(\mathsf{P}_{\mathsf{lam}\,t}\,(\pi_1\,\rho, \mathsf{proj}_1\,\alpha)\big)\,\mathsf{id}\,(\pi_2\,\rho)\,(\mathsf{proj}_2\,\alpha) \\
&= \mathsf{P}_t\,\Big(\big(\mathsf{TM}_\Gamma\,\mathsf{id}\,(\pi_1\,\rho), \pi_2\,\rho\big), \big(\mathsf{P}_\Gamma\,\mathsf{id}\,(\mathsf{proj}_1\,\alpha), \mathsf{proj}_2\,\alpha\big)\Big) \\
&\hspace{16em} (\text{identity law for } \mathsf{TM}_\Gamma \text{ and } \mathsf{P}_\Gamma) \\
&\equiv \mathsf{P}_t\,\big((\pi_1\,\rho, \pi_2\,\rho), (\mathsf{proj}_1\,\alpha, \mathsf{proj}_2\,\alpha)\big) \\
&\hspace{12em} (\eta \text{ law for substitutions and metatheoretic } \Sigma) \\
&\equiv \mathsf{P}_t\,(\rho, \alpha)
\end{aligned}
$$

For $\Pi\eta$ we only compare the map components as the naturality components are equal by K.

$$
\begin{aligned}
&\mathsf{map}\,\big(\mathsf{P}_{\mathsf{lam}\,(\mathsf{app}\,t)}\,(\rho, \alpha)\big) \\
&= \lambda\beta\,u\,v.\mathsf{P}_{\mathsf{app}\,t}\,\big((\mathsf{TM}_\Gamma\,\beta\,\rho, u), (\mathsf{P}_\Gamma\,\beta\,\alpha, v)\big) \\
&= \lambda\beta\,u\,v.\mathsf{map}\,\big(\mathsf{P}_t\,(\pi_1\,(\mathsf{TM}_\Gamma\,\beta\,\rho, u), \mathsf{P}_\Gamma\,\beta\,\alpha)\big)\,\mathsf{id}\,\big(\pi_2\,(\mathsf{TM}_\Gamma\,\beta\,\rho, u)\big)\,v \\
&\hspace{20em} (\text{substitution calculus})
\end{aligned}
$$

$$
\begin{aligned}
&\equiv \lambda\beta\, u\, v.\mathsf{map}\,\big(\mathsf{P}_t\,(\mathsf{TM}_\Gamma\,\beta\,\rho, \mathsf{P}_\Gamma\,\beta\,\alpha)\big)\,\mathsf{id}\,u\,v \\
&\hfill (\text{naturality of } \mathsf{P}_t) \\
&\equiv \lambda\beta\, u\, v.\mathsf{map}\,\big(\mathsf{P}_{\Pi\,A\,B}\,\beta\,(\mathsf{P}_t\,(\rho,\alpha))\big)\,\mathsf{id}\,u\,v \\
&\hfill (\mathsf{map}\circ) \\
&\equiv \lambda\beta\, u\, v.\mathsf{map}\,\big(\mathsf{P}_t\,(\rho,\alpha)\big)\,(\beta\circ\mathsf{id})\,u\,v \\
&\hfill (\circ\mathsf{id}) \\
&\equiv \lambda\beta\, u\, v.\mathsf{map}\,\big(\mathsf{P}_t\,(\rho,\alpha)\big)\,\beta\,u\,v \\
&= \mathsf{map}\,\big(\mathsf{P}_t\,(\rho,\alpha)\big)
\end{aligned}
$$

Finally, we have to verify lam[]. As before, we only verify the map components.

$$
\begin{aligned}
&\mathsf{map}\,\big(\mathsf{P}_{(\mathsf{lam}\,t)[\sigma]}\,(\rho,\alpha)\big) \\
&= \mathsf{map}\,\Big(\mathsf{P}_{\mathsf{lam}\,t}\,\big(\mathsf{TM}_\sigma\,\rho, \mathsf{P}_\sigma\,(\rho,\alpha)\big)\Big) \\
&= \lambda\beta\, u\, v.\mathsf{P}_t\,\Big(\big(\mathsf{TM}_\Gamma\,\beta\,(\mathsf{TM}_\sigma\,\rho), u\big), \big(\mathsf{P}_\Gamma\,\beta\,(\mathsf{P}_\sigma\,(\rho,\alpha)), v\big)\Big) \\
&\hfill (\text{naturality of } \mathsf{TM}_\sigma \text{ and } \mathsf{P}_\sigma) \\
&\equiv \lambda\beta\, u\, v.\mathsf{P}_t\,\Big(\big(\mathsf{TM}_\sigma\,(\mathsf{TM}_\Gamma\,\beta\,\rho), u\big), \big(\mathsf{P}_\sigma\,(\mathsf{TM}_\Gamma\,\beta\,\rho, \mathsf{P}_\Gamma\,\beta\,\alpha), v\big)\Big) \\
&\hfill (\text{substitution calculus and } \mathsf{P}_\uparrow) \\
&\equiv \lambda\beta\, u\, v.\mathsf{P}_t\,\Big(\mathsf{TM}_{\sigma\uparrow}\,(\mathsf{TM}_\Gamma\,\beta\,\rho, u), \mathsf{P}_{\sigma\uparrow}\,\big((\mathsf{TM}_\Gamma\,\beta\,\rho, u), (\mathsf{P}_\Gamma\,\beta\,\alpha, v)\big)\Big) \\
&= \lambda\beta\, u\, v.\mathsf{P}_{t[\sigma\uparrow]}\,\big((\mathsf{TM}_\Gamma\,\beta\,\rho, u), (\mathsf{P}_\Gamma\,\beta\,\alpha, v)\big) \\
&= \mathsf{map}\,\big(\mathsf{P}_{\mathsf{lam}\,t[\sigma\uparrow]}\,(\rho,\alpha)\big)
\end{aligned}
$$

This concludes the definition of the logical predicate interpretation.

5.8 Normal forms

In this section we define the Yoneda embedding for normal terms analogously to TM. For a type $\Gamma \vdash A$ we will denote it by $\mathsf{NF}_A : \mathsf{FamPSh}\,\mathsf{TM}_\Gamma$. We also define another family $\mathsf{NF}^{\equiv}{}_A : \mathsf{FamPSh}\,(\Sigma\,\mathsf{TM}_\Gamma\,\mathsf{TM}_A)$ which expresses that there exists a normal form in NF_A that is equal to the term in TM_A.

As a first step, we define η-long β-normal forms mutually with neutral terms and the embedding $\ulcorner - \urcorner$ back to terms. These are the normal forms for the syntax with the substitution calculus, base type, base family and function space. Note that neutral terms and normal forms are indexed by types, not normal types. Variables are the ones defined in section 5.6.

$$
\begin{aligned}
&\mathsf{data}\,\mathsf{Ne} && : (\Gamma : \mathsf{Con}) \to \mathsf{Ty}\,\Gamma \to \mathsf{Set}\\
&\mathsf{data}\,\mathsf{Nf} && : (\Gamma : \mathsf{Con}) \to \mathsf{Ty}\,\Gamma \to \mathsf{Set}\\
&\ulcorner - \urcorner && : \mathsf{Nf}\,\Gamma\,A \to \mathsf{Tm}\,\Gamma\,A\\
&\mathsf{data}\,\mathsf{Ne} &&\\
&\quad\mathsf{var} && : \mathsf{Var}\,\Gamma\,A \to \mathsf{Ne}\,\Gamma\,A\\
&\quad\mathsf{app} && : \mathsf{Ne}\,\Gamma\,(\Pi\,A\,B) \to (v : \mathsf{Nf}\,\Gamma\,A) \to \mathsf{Ne}\,\Gamma\,(B[\langle \ulcorner v \urcorner \rangle])\\
&\mathsf{data}\,\mathsf{Nf} &&\\
&\quad\mathsf{neuU} && : \mathsf{Ne}\,\Gamma\,\mathsf{U} \to \mathsf{Nf}\,\Gamma\,\mathsf{U}\\
&\quad\mathsf{neuEl} && : \mathsf{Ne}\,\Gamma\,(\mathsf{El}\,\hat{A}) \to \mathsf{Nf}\,\Gamma\,(\mathsf{El}\,\hat{A})\\
&\quad\mathsf{lam} && : \mathsf{Nf}\,(\Gamma, A)\,B \to \mathsf{Nf}\,\Gamma\,(\Pi\,A\,B)\\
&\ulcorner - \urcorner && : \mathsf{Ne}\,\Gamma\,A \to \mathsf{Tm}\,\Gamma\,A
\end{aligned}
$$

Neutral terms are terms where a variable is in a key position which precludes the application of the rule $\Pi\beta$. η-long normal forms mean that only neutral terms of the base types are normal forms. This is required in order to have no redundancy in normal forms. E.g. we don't want to embed a neutral term $n : \mathsf{Ne}\,\Gamma\,(\Pi\,A\,\mathsf{U})$ into normal forms, only its η-expanded variant: first we do $\mathsf{app}\,n\,(\mathsf{var}\,\mathsf{vz}) : \mathsf{Ne}\,(\Gamma, A)\,\mathsf{U}$ and now we can embed it into normal forms by $\mathsf{lam}\,\big(\mathsf{app}\,n\,(\mathsf{var}\,\mathsf{vz})\big) : \mathsf{Nf}\,\Gamma\,(\Pi\,A\,\mathsf{U})$.

The embeddings into terms are defined in the obvious way.

$$
\begin{aligned}
&\ulcorner \mathsf{var}\,x \urcorner && := \ulcorner x \urcorner\\
&\ulcorner \mathsf{app}\,v\,n \urcorner && := \ulcorner v \urcorner \$ \ulcorner n \urcorner\\
&\ulcorner \mathsf{neuU}\,n \urcorner && := \ulcorner n \urcorner\\
&\ulcorner \mathsf{neuEl}\,n \urcorner && := \ulcorner n \urcorner\\
&\ulcorner \mathsf{lam}\,v \urcorner && := \mathsf{lam}\,\ulcorner v \urcorner
\end{aligned}
$$

We define lists of neutral terms and normal forms. X is a parameter of the list, it can stand for both Ne and Nf.

$$
\begin{aligned}
&\mathsf{data}\,{-}\mathsf{s}\,(X : (\Gamma : \mathsf{Con}) \to \mathsf{Ty}\,\Gamma \to \mathsf{Set}) : \mathsf{Con} \to \mathsf{Con} \to \mathsf{Set}\\
&\ulcorner - \urcorner : X\mathsf{s}\,\Gamma\,\Delta \to \mathsf{Tms}\,\Gamma\,\Delta
\end{aligned}
$$

$$\begin{aligned}
&\mathsf{data}\, X\mathsf{s} \\
&\quad \epsilon \quad\;\; : X\mathsf{s}\,\Gamma\,\cdot \\
&\quad -,- : (\tau : X\mathsf{s}\,\Gamma\,\Delta) \to X\,\Gamma\,A[\ulcorner\tau\urcorner] \to X\mathsf{s}\,\Gamma\,(\Delta, A)
\end{aligned}$$

The embeddings into substitutions are defined pointwise.

$$\begin{aligned}
\ulcorner\epsilon\urcorner \quad &:= \epsilon \\
\ulcorner\tau, n\urcorner &:= \ulcorner\tau\urcorner, \ulcorner n\urcorner
\end{aligned}$$

We also need renamings of (lists of) normal forms and neutral terms together with lemmas relating their embeddings to terms. Again, X can stand for both Ne and Nf.

$$\begin{aligned}
-[-] \;\;&: X\,\Gamma\,A \to (\beta : \mathsf{Vars}\,\Psi\,\Gamma) \to X\,\Psi\,A[\ulcorner\beta\urcorner] & \ulcorner[]\urcorner : \ulcorner n\urcorner[\ulcorner\beta\urcorner] \equiv \ulcorner n[\beta]\urcorner \\
- \circ - &: X\mathsf{s}\,\Gamma\,\Delta \to \mathsf{Vars}\,\Psi\,\Gamma \to X\mathsf{s}\,\Psi\,\Delta & \ulcorner\circ\urcorner : \ulcorner\tau\urcorner \circ \ulcorner\beta\urcorner \equiv \ulcorner\tau \circ \beta\urcorner
\end{aligned}$$

These functions are all defined by induction on the first argument. Note that in the case of lam we need to lift the renaming so that it leaves the last component of the context untouched.

$$\begin{aligned}
(\mathsf{var}\, x)[\beta] \quad &:= \mathsf{var}\,(x[\beta]) \\
(\mathsf{app}\, v\, n)[\beta] &:= \mathsf{app}\,(v[\beta])\,(n[\beta]) \\
(\mathsf{neuU}\, n)[\beta] &:= \mathsf{neuU}\,(n[\beta]) \\
(\mathsf{neuEl}\, n)[\beta] &:= \mathsf{neuEl}\,(n[\beta]) \\
(\mathsf{lam}\, v)[\beta] \quad &:= \mathsf{lam}\,(v[\beta \circ \mathsf{wk}, \mathsf{vz}]) \\
\epsilon \circ \beta \qquad &:= \epsilon \\
(\tau, n) \circ \beta \quad &:= (\tau \circ \beta, {}_{\ulcorner\circ\urcorner *} n[\beta])
\end{aligned}$$

In addition, we prove the functor laws for both (lists of) neutral terms and normal forms by induction on n (and τ, respectively).

$$\begin{aligned}
n[\mathsf{id}] &\equiv n & \tau \circ \mathsf{id} &\equiv \tau \\
n[\beta][\gamma] &\equiv n[\beta \circ \gamma] & (\tau \circ \beta) \circ \gamma &\equiv \tau \circ (\beta \circ \gamma)
\end{aligned}$$

Now we can define the presheaf X_Γ and families of presheaves X_A for any

$A : \mathsf{Ty}\,\Gamma$ where X is either NE or NF. The definitions follow those of TM.

$$\begin{array}{llll}
\Delta : \mathsf{Con} & X_\Delta : \mathsf{PSh}\,\mathsf{REN} & X_\Delta\,\Psi \;:= X\mathsf{s}\,\Psi\,\Delta & X_\Delta\,\beta\,\tau := \tau \circ \beta \\
A : \mathsf{Ty}\,\Gamma & X_A : \mathsf{FamPSh}\,\mathsf{TM}_\Gamma & X_{A\,\Psi}\,\rho := X\,\Psi\,A[\rho] & X_A\,\beta\,n := n[\beta]
\end{array}$$

Functoriality is given by the above mentioned functor laws.

We define the natural transformation which embeds neutral substitutions into substitutions and the family of natural transformations which embeds neutral terms into terms.

$$\ulcorner - \urcorner : \mathsf{NE}_\Delta \dot{\to} \mathsf{TM}_\Delta \qquad\qquad \ulcorner - \urcorner : \mathsf{NE}_A \overset{\mathsf{N}}{\to} \mathsf{TM}_A$$

They are simply defined by $\ulcorner - \urcorner$ for (lists of) neutral terms and naturality is given by $\ulcorner \circ \urcorner$ and $\ulcorner [] \urcorner$.

We define a family of presheaves over TM_Δ expressing that substitutions have a normal form which is equal to them.

$$\begin{array}{ll}
\mathsf{NF}^{\equiv}{}_{(\Delta:\mathsf{Con})} & : \mathsf{FamPSh}\,\mathsf{TM}_\Delta \\
\mathsf{NF}^{\equiv}{}_\Delta\,(\rho : \mathsf{TM}_\Delta\,\Psi) & := \Sigma(\rho' : \mathsf{NF}_\Delta\,\Psi).\rho \equiv \ulcorner \rho' \urcorner \\
\mathsf{NF}^{\equiv}{}_\Delta\,(\beta : \mathsf{Vars}\,\Omega\,\Psi)\,(\rho', p) & : \underbrace{\mathsf{NF}^{\equiv}{}_\Delta\,(\mathsf{TM}_\Delta\,\beta\,\rho)}_{=\Sigma(\rho':\mathsf{NF}_\Delta\,\Omega).\rho\circ\ulcorner\beta\urcorner\equiv\ulcorner\rho'\urcorner} \\
& := \mathsf{NF}_\Delta\,\beta\,\rho', \mathsf{ap}\,(\mathsf{TM}_\Delta\,\ulcorner\beta\urcorner)\,p \centerdot \ulcorner\circ\urcorner
\end{array}$$

The functor laws are satisfied by

$$\mathsf{NF}^{\equiv}{}_\Delta\,\mathsf{id}\,(\rho', p) = (\rho' \circ \mathsf{id}), \mathsf{ap}\,(- \circ \ulcorner \mathsf{id} \urcorner)\,p \centerdot \ulcorner\circ\urcorner \equiv (\rho', p)$$

and

$$\begin{aligned}
&\mathsf{NF}^{\equiv}{}_\Delta\,(\beta \circ \gamma)\,(\rho', p) \\
=\,&\rho' \circ (\beta \circ \gamma), \mathsf{ap}\,(- \circ \ulcorner \beta \circ \gamma \urcorner)\,p \centerdot \ulcorner\circ\urcorner \\
&& (\circ\circ, \ulcorner\circ\urcorner, \mathsf{K}) \\
\equiv\,&\rho' \circ (\beta \circ \gamma), \mathsf{ap}\,\big((- \circ \ulcorner\beta\urcorner) \circ \ulcorner\gamma\urcorner\big)\,p \centerdot \mathsf{ap}\,(- \circ \ulcorner\gamma\urcorner)\,\ulcorner\circ\urcorner \centerdot \ulcorner\circ\urcorner \\
&& (\mathsf{apap}) \\
\equiv\,&\rho' \circ (\beta \circ \gamma), \mathsf{ap}\,(- \circ \ulcorner\gamma\urcorner)\,\big(\mathsf{ap}\,(- \circ \ulcorner\beta\urcorner)\,p\big) \centerdot \mathsf{ap}\,(- \circ \ulcorner\gamma\urcorner)\,\ulcorner\circ\urcorner \centerdot \ulcorner\circ\urcorner
\end{aligned}$$

$$\begin{aligned}
& && (\circ\circ, \mathsf{ap}{\scriptscriptstyle\blacksquare}) \\
&\equiv (\rho' \circ \beta) \circ \gamma, \mathsf{ap}\,(- \circ \ulcorner \gamma \urcorner)\,\big(\mathsf{ap}\,(- \circ \ulcorner \beta \urcorner)\,p \mathbin{\scriptscriptstyle\blacksquare} \ulcorner \circ \urcorner\big) \mathbin{\scriptscriptstyle\blacksquare} \ulcorner \circ \urcorner \\
&= \mathsf{NF}^{\equiv}{}_{\Delta}\,\gamma\,\big(\mathsf{NF}^{\equiv}{}_{\Delta}\,\beta\,(\rho', p)\big)
\end{aligned}$$

Analogously, we define $\mathsf{NF}^{\equiv}$ for terms.

$$\begin{aligned}
&\mathsf{NF}^{\equiv}{}_{(A:\mathsf{Ty}\,\Gamma)} && : \mathsf{FamPSh}\,(\Sigma\,\mathsf{TM}_{\Gamma}\,\mathsf{TM}_{A}) \\
&\mathsf{NF}^{\equiv}{}_{A}\,(\rho : \mathsf{TM}_{\Gamma}\,\Psi, s : \mathsf{TM}_{A}\,\rho) && := \Sigma(s' : \mathsf{NF}_{A}\,\rho).s \equiv \ulcorner s' \urcorner \\
&\mathsf{NF}^{\equiv}{}_{A}\,(\beta : \mathsf{Vars}\,\Omega\,\Psi)\,(s', p) && : \underbrace{\mathsf{NF}^{\equiv}{}_{A}\,(\mathsf{TM}_{\Gamma}\,\beta\,\rho, \mathsf{TM}_{A}\,\beta\,s)}_{=\Sigma(s':\mathsf{NF}_{A}\,(\mathsf{TM}_{\Gamma}\,\beta\,\rho)).s[\ulcorner\beta\urcorner]\equiv\ulcorner s'\urcorner} \\
& && := \mathsf{NF}_{A}\,\beta\,s', \mathsf{ap}\,(\mathsf{TM}_{A}\,\ulcorner \beta \urcorner)\,p \mathbin{\scriptscriptstyle\blacksquare} \ulcorner [] \urcorner
\end{aligned}$$

The functor laws can be proven the same way as above.

$$\begin{aligned}
&\mathsf{NF}^{\equiv}{}_{A}\,\mathsf{id}\,(s', p) = (s'[\mathsf{id}]), \mathsf{ap}\,(-[\ulcorner \mathsf{id} \urcorner])\,p \mathbin{\scriptscriptstyle\blacksquare} \ulcorner [] \urcorner \equiv (s', p) \\
&\mathsf{NF}^{\equiv}{}_{A}\,(\beta \circ \gamma)\,(s', p) \\
&= s'[\beta \circ \gamma], \mathsf{ap}\,(-[\ulcorner \beta \circ \gamma \urcorner])\,p \mathbin{\scriptscriptstyle\blacksquare} \ulcorner [] \urcorner \\
& && ([][], \ulcorner [] \urcorner, \mathsf{apap}, \mathsf{ap}{\scriptscriptstyle\blacksquare}, \mathsf{K}) \\
&\equiv s'[\beta][\gamma], \mathsf{ap}\,(-[\gamma])\,\big(\mathsf{ap}\,(-[\beta])\,p \mathbin{\scriptscriptstyle\blacksquare} \ulcorner [] \urcorner\big) \mathbin{\scriptscriptstyle\blacksquare} \ulcorner [] \urcorner \\
&= \mathsf{NF}^{\equiv}{}_{A}\,\gamma\,\big(\mathsf{NF}^{\equiv}{}_{A}\,\beta\,(s', p)\big)
\end{aligned}$$

We set the parameters of the logical predicate at the base type and family by defining $\bar{\mathsf{U}}$ and $\bar{\mathsf{El}}$. We will use $\mathsf{NF}^{\equiv}$ to define them. The predicate at the base type for a term says that there exists a normal form at the base type which is equal to this term.

$$\begin{aligned}
&\bar{\mathsf{U}} : \mathsf{FamPSh}\,\mathsf{TM}^{\mathsf{U}} \\
&\bar{\mathsf{U}}_{\Psi}\,(\hat{A} : \mathsf{Tm}\,\Psi\,\mathsf{U}) := \mathsf{NF}^{\equiv}{}_{\mathsf{U}}\,(\mathsf{id}, {}_{[\mathsf{id}]*}\hat{A}) \\
&\bar{\mathsf{U}}\,(\beta : \mathsf{Vars}\,\Omega\,\Psi)\,\big(\alpha : \mathsf{NF}^{\equiv}{}_{\mathsf{U}}\,(\mathsf{id}, \hat{A})\big) : \mathsf{NF}^{\equiv}{}_{\mathsf{U}}\,(\mathsf{id}, \hat{A}[\ulcorner \beta \urcorner]) := {}_{\mathsf{U}[]*}\mathsf{NF}^{\equiv}\,\beta\,\alpha
\end{aligned}$$

Functoriality follows from that of $\mathsf{NF}^{\equiv}{}_{\mathsf{U}}$.

$$\begin{aligned}
&\bar{\mathsf{El}} : \mathsf{FamPSh}\,\Big(\Sigma\,\big(\Sigma\,(\mathsf{TM}^{\mathsf{U}}\,\mathsf{TM}^{\mathsf{El}})\big)\,\bar{\mathsf{U}}[\mathsf{wk}]\Big) \\
&\bar{\mathsf{El}}_{\Psi}\,(\hat{A}, t : \mathsf{Tm}\,\Psi\,(\mathsf{El}\,\hat{A}), p) := \mathsf{NF}^{\equiv}{}_{\mathsf{El}\,\hat{A}}\,(\mathsf{id}, {}_{[\mathsf{id}]*}t)
\end{aligned}$$

$$\bar{\mathsf{El}}\,(\beta : \mathsf{Vars}\,\Omega\,\Psi)\,\big(\alpha : \mathsf{NF}^{\equiv}{}_{\mathsf{El}\,\hat{A}}\,(\mathsf{id}, t)\big) : \mathsf{NF}^{\equiv}{}_{\mathsf{El}\,\hat{A}}\,(\mathsf{id}, t[\ulcorner\beta\urcorner]) := {}_{\mathsf{El}[]*}\mathsf{NF}^{\equiv}\,\beta\,\alpha$$

Functoriality follows from that of $\mathsf{NF}^{\equiv}{}_{\mathsf{El}\,\hat{A}}$.

Now we can interpret any term in the logical predicate interpretation over REN with base type interpretations $\bar{\mathsf{U}}$ and $\bar{\mathsf{El}}$. We denote the interpretation of t by P_t.

Note that we can embed renamings into neutral substitutions pointwise.

5.9 Quote and unquote

5.9.1 Motives

By the logical predicate interpretation parameterised by $\bar{\mathsf{U}}$ and $\bar{\mathsf{El}}$ we have the following two things:

- a term at the base type or base family is equal to a normal form,
- this property is preserved by the other type formers (functions and substituted types).

We make use of this fact to lift the first property to any type. We do this by defining a quote function by induction on the type. Quote takes a term which preserves the predicate and maps it to a normal form that it is equal to it. Because of the nature of function space, we need an unquote function in the other direction as well, mapping neutral terms to the witness of the predicate.

More precisely, we define the quote function q and unquote u by induction on the structure of contexts and types. For this, we need to define a model of type theory in which only the motives for contexts and types are interesting.

The motives for a context Δ and a type $A : \mathsf{Ty}\,\Gamma$ are given below and are also depicted in figure 5.4. To express the commutativity of the diagrams we use sections and the $\mathsf{NF}^{\equiv}$ families of presheaves. We only write Σ once for iterated usage.

$$\begin{aligned}
\mathsf{u}_\Delta &: \mathsf{NE}_\Delta && \xrightarrow{\mathsf{s}} \mathsf{P}_\Delta[\ulcorner - \urcorner] \\
\mathsf{q}_\Delta &: \Sigma\,\mathsf{TM}_\Delta\,\mathsf{P}_\Delta && \xrightarrow{\mathsf{s}} \mathsf{NF}^{\equiv}{}_\Delta[\mathsf{wk}] \\
\mathsf{u}_A &: \Sigma\,\mathsf{TM}_\Gamma\,\mathsf{NE}_A\,\mathsf{P}_\Gamma[\mathsf{wk}] && \xrightarrow{\mathsf{s}} \mathsf{P}_A[\mathsf{id}, \ulcorner - \urcorner, \mathsf{id}] \\
\mathsf{q}_A &: \Sigma\,\mathsf{TM}_\Gamma\,\mathsf{TM}_A\,\mathsf{P}_\Gamma[\mathsf{wk}]\,\mathsf{P}_A && \xrightarrow{\mathsf{s}} \mathsf{NF}^{\equiv}{}_A[\mathsf{wk}][\mathsf{wk}]
\end{aligned}$$

$$\begin{tikzcd}
\mathsf{NE}_\Delta \arrow[r, "\mathsf{u}_\Delta"] \arrow[dr, "\ulcorner\_\urcorner"'] & \Sigma\,\mathsf{TM}_\Delta\,\mathsf{P}_\Delta \arrow[r, "\mathsf{q}_\Delta"] \arrow[d, "\mathsf{proj}"] & \mathsf{NF}_\Delta \arrow[dl, "\ulcorner\_\urcorner"] \\
& \mathsf{TM}_\Delta &
\end{tikzcd}$$

$$\begin{tikzcd}
\Sigma\,\mathsf{TM}_\Gamma\,\mathsf{NE}_A\,\mathsf{P}_\Gamma[\mathsf{wk}] \arrow[r, "\mathsf{u}_A"] \arrow[dr, "\ulcorner\_\urcorner"'] & \Sigma\,\mathsf{TM}_\Gamma\,\mathsf{TM}_A\,\mathsf{P}_\Gamma[\mathsf{wk}]\,\mathsf{P}_A \arrow[r, "\mathsf{q}_A"] \arrow[d, "\mathsf{proj}"] & \Sigma\,\mathsf{TM}_\Gamma\,\mathsf{NF}_A\,\mathsf{P}_\Gamma[\mathsf{wk}] \arrow[dl, "\ulcorner\_\urcorner"] \\
& \Sigma\,\mathsf{TM}_\Gamma\,\mathsf{TM}_A\,\mathsf{P}_\Gamma[\mathsf{wk}] &
\end{tikzcd}$$

Figure 5.4: The type of quote and unquote for a context Δ and a type $\Gamma \vdash A$. We only write one Σ for iterated Σs.

In the type of u_A we define the natural transformation $\mathsf{id}, \ulcorner - \urcorner, \mathsf{id}$ in the obvious way, it just embeds the second components (neutral terms) into terms.

Unquote for a context takes a neutral substitution and returns a proof that the logical predicate holds for it. Quote takes a substitution for which the predicate holds and returns a normal substitution together with a proof of convertibility. The type of unquote and quote for types is more involved as they depend on a substitution for which the predicate needs to hold.

The motives for substitutions and terms are just the constant $\top$ functions, hence all their methods are trivial.

5.9.2 Methods for the substitution calculus

Unqoute and quote for the empty context return the element of the unit type and the empty substitution. Naturality is trivial in both cases.

$$\begin{aligned}
&\mathsf{u}.\,(\tau : \mathsf{NE}.\,\Psi) && : \top && := \mathsf{tt} \\
&\mathsf{q}.\,\big((\sigma : \mathsf{TM}.\,\Psi), (\alpha : \top)\big) && : \Sigma(\rho' : \mathsf{NF}.\,\Psi).\rho \equiv \ulcorner \rho' \urcorner && := (\epsilon, \epsilon\eta)
\end{aligned}$$

For extended contexts, unquote and quote just call the corresponding functions for each type pointwise. As the unqoute for a type depends on the witness of the predicate for the context, we need to use $\mathsf{u}_\Delta\,\tau$ again.

$$\begin{aligned}\mathsf{u}_{\Delta,A}\,\big((\tau,n) : \mathsf{NE}_{\Delta,A}\,\Psi\big) &: \Sigma\big(\alpha : \mathsf{P}_{\Delta}\,(\pi_1\,\ulcorner\tau,n\urcorner)\big).\mathsf{P}_{A}\,(\pi_1\,\ulcorner\tau,n\urcorner,\pi_2\,\ulcorner\tau,n\urcorner,\alpha)\\ &:= \mathsf{u}_{\Delta}\,\tau,\mathsf{u}_{A}\,(\ulcorner\tau\urcorner,n,\mathsf{u}_{\Delta}\,\tau)\end{aligned}$$

To define quote pointwise, we need to split the substitution ρ into two by using π_1 and π_2. Then we split the results again using let bindings because we need to put the normal forms and the equalities back pointwise again. ap, is the congruence law for substitution extension (see section 3.3).

$$\begin{aligned}\mathsf{q}_{\Delta,A}\,\big((\rho : \mathsf{TM}_{\Delta}\,\Psi),(\alpha,a) : \mathsf{P}_{\Delta,A}\,\rho\big) &: \Sigma(\rho' : \mathsf{NF}_{\Delta,A}\,\Psi).\rho\equiv\ulcorner\rho'\urcorner\\ &:= \mathsf{let}\,(\tau,p) := \mathsf{q}_{\Delta}\,(\pi_1\,\rho,\alpha)\\ &\qquad\quad (n,q) := \mathsf{q}_{A}\,(\pi_1\,\rho,\pi_2\,\rho,\alpha,a)\\ &\qquad\ \mathsf{in}\ \ (\tau,n),(\mathsf{ap},p\,q)\end{aligned}$$

Naturality is proven pointwise.

$$\begin{aligned}&\mathsf{P}_{\Delta,A}\,\beta\,\big(\mathsf{u}_{\Delta,A}\,(\tau,n)\big)\\ =\,&\mathsf{P}_{\Delta}\,\beta\,(\mathsf{u}_{\Delta}\,\tau),\mathsf{P}_{A}\,\beta\,\big(\mathsf{u}_{A}\,(\ulcorner\tau\urcorner,n,\mathsf{u}_{\Delta}\,\tau)\big)\\ &\hfill\text{(naturality of } \mathsf{u}_{\Delta} \text{ and } \mathsf{u}_{A}\text{)}\\ \equiv\,&\mathsf{u}_{\Delta}\,(\mathsf{NE}_{\Delta}\,\beta\,\tau),\mathsf{u}_{A}\,\big(\mathsf{TM}_{\Delta}\,\beta\,\ulcorner\tau\urcorner,\mathsf{NE}_{A}\,\beta\,n,\mathsf{u}_{\Delta}\,(\mathsf{NE}_{\Delta}\,\beta\,\tau)\big)\\ &\hfill\text{(naturality of the embedding into terms } \ulcorner-\urcorner\text{)}\\ \equiv\,&\mathsf{u}_{\Delta}\,(\mathsf{NE}_{\Delta}\,\beta\,\tau),\mathsf{u}_{A}\,\big(\ulcorner\mathsf{NE}_{\Delta}\,\beta\,\tau\urcorner,\mathsf{NE}_{A}\,\beta\,n,\mathsf{u}_{\Delta}\,(\mathsf{NE}_{\Delta}\,\beta\,\tau)\big)\\ =\,&\mathsf{u}_{\Delta,A}\,\big(\mathsf{NE}_{\Delta,A}\,\beta\,(\tau,n)\big)\end{aligned}$$

To prove naturality for quote, it is enough to prove the NF part of $\mathsf{NF}^{\equiv}$ as the equalities will be equal by K.

$$\begin{aligned}&\mathsf{NF}_{\Delta,A}\,\beta\,\Big(\mathsf{proj}_1\,\big(\mathsf{q}_{\Delta,A}\,(\rho,(\alpha,a))\big)\Big)\\ =\ \ &\mathsf{NF}_{\Delta}\,\beta\,\Big(\mathsf{proj}_1\,\big(\mathsf{q}_{\Delta}\,(\pi_1\,\rho,\alpha)\big)\Big)\\ &,\mathsf{NF}_{A}\,\beta\,\Big(\mathsf{proj}_1\,\big(\mathsf{q}_{A}\,(\pi_1\,\rho,\pi_2\,\rho,\alpha,a)\big)\Big)\\ &\hfill\text{(naturality of } \mathsf{q}_{\Delta} \text{ and } \mathsf{q}_{A}\text{)}\\ \equiv\ \ &\mathsf{proj}_1\,\big(\mathsf{q}_{\Delta}\,(\mathsf{TM}_{\Delta}\,\beta\,(\pi_1\,\rho),\mathsf{P}_{\Delta}\,\beta\,\alpha)\big)\end{aligned}$$

$$
\begin{aligned}
&\quad , \mathsf{proj}_1 \big(\mathsf{q}_A \, (\mathsf{TM}_\Delta \, \beta \, (\pi_1 \, \rho), \mathsf{TM}_A \, \beta \, (\pi_2 \, \rho), \mathsf{P}_\Delta \, \beta \, \alpha, \mathsf{P}_A \, \beta \, a)\big) \\
&\hspace{20em} (\pi_1\circ, \pi_2[]) \\
&\equiv \ \mathsf{proj}_1 \big(\mathsf{q}_\Delta \, (\pi_1 \, (\mathsf{TM}_{\Delta\, A} \, \beta \, \rho), \mathsf{P}_\Delta \, \beta \, \alpha)\big) \\
&\quad , \mathsf{proj}_1 \big(\mathsf{q}_A \, (\pi_1 \, (\mathsf{TM}_{\Delta, A} \, \beta \, \rho), \pi_2 \, (\mathsf{TM}_{\Delta, A} \, \beta \, \rho), \mathsf{P}_\Delta \, \beta \, \alpha, \mathsf{P}_A \, \beta \, a)\big) \\
&= \mathsf{proj}_1 \Big(\mathsf{q}_{\Delta, A} \, \big(\mathsf{TM}_{\Delta\, A} \, \beta \, \rho, \mathsf{P}_{\Delta, A} \, \beta \, (\alpha, a)\big)\Big)
\end{aligned}
$$

Substituted types are quoted and unquoted at the substituted inputs.

$$
\begin{aligned}
\mathsf{u}_{A[\sigma]} \, (\rho, n, \alpha) \quad &: \mathsf{P}_A \, \big(\mathsf{TM}_\sigma \, \rho, \ulcorner n \urcorner, \mathsf{P}_\sigma \, (\rho, \alpha)\big) := \mathsf{u}_A \, \big(\mathsf{TM}_\sigma \, \rho, n, \mathsf{P}_\sigma \, (\rho, \alpha)\big) \\
\mathsf{q}_{A[\sigma]} \, (\rho, s, \alpha, a) &: \Sigma(s' : \mathsf{NF}_{A[\sigma]} \, \rho).s \equiv \ulcorner s' \urcorner \quad := \mathsf{q}_A \, \big(\mathsf{TM}_\sigma \, \rho, s, \mathsf{P}_\sigma \, (\rho, \alpha), a\big)
\end{aligned}
$$

Naturality for $A[\sigma]$ follows from naturality of A and TM_σ and P_σ.

$$
\begin{aligned}
&\quad \mathsf{P}_{A[\sigma]} \, \beta \, \big(\mathsf{u}_{A[\sigma]} \, (\rho, n, \alpha)\big) \\
&= \mathsf{P}_A \, \beta \, \Big(\mathsf{u}_A \, \big(\mathsf{TM}_\sigma \, \rho, n, \mathsf{P}_\sigma \, (\rho, \alpha)\big)\Big) \\
&\hspace{20em} (\text{naturality of } \mathsf{u}_A) \\
&\equiv \mathsf{u}_A \, \big(\mathsf{TM}_\Theta \, \beta \, (\mathsf{TM}_\sigma \, \rho), \mathsf{NE}_A \, \beta \, n, \mathsf{P}_\Theta \, \beta \, (\mathsf{P}_\sigma \, (\rho, \alpha))\big) \\
&\hspace{16em} (\text{naturality of } \mathsf{TM}_\sigma \text{ and } \mathsf{P}_\sigma) \\
&\equiv \mathsf{u}_A \, \big(\mathsf{TM}_\sigma \, (\mathsf{TM}_\Gamma \, \beta \, \rho), \mathsf{NE}_{A[\sigma]} \, \beta \, n, \mathsf{P}_\sigma \, (\mathsf{TM}_\Gamma \, \beta \, \rho, \mathsf{P}_\Gamma \, \beta \, \alpha)\big) \\
&= \mathsf{u}_{A[\sigma]} \, (\mathsf{TM}_\Gamma \, \beta \, \rho, \mathsf{NE}_{A[\sigma]} \, \beta \, n, \mathsf{P}_\Gamma \, \beta \, \alpha)
\end{aligned}
$$

$$
\begin{aligned}
&\quad \mathsf{NF}_{A[\sigma]} \, \beta \, \Big(\mathsf{proj}_1 \, \big(\mathsf{q}_{A[\sigma]} \, (\rho, s, \alpha, a)\big)\Big) \\
&= \mathsf{NF}_A \, \beta \, \Big(\mathsf{proj}_1 \big(\mathsf{q}_A \, \big(\mathsf{TM}_\sigma \, \rho, s, \mathsf{P}_\sigma \, (\rho, \alpha), a\big)\big)\Big) \\
&\hspace{20em} (\text{naturality of } \mathsf{q}_A) \\
&\equiv \mathsf{proj}_1 \, \Big(\mathsf{q}_A \, \big(\mathsf{TM}_\Theta \, \beta \, (\mathsf{TM}_\sigma \, \rho), \mathsf{TM}_A \, \beta \, s, \mathsf{P}_\Theta \, \beta \, (\mathsf{P}_\sigma \, (\rho, \alpha)), \mathsf{P}_A \, \beta \, a\big)\Big) \\
&\hspace{16em} (\text{naturality of } \mathsf{TM}_\sigma \text{ and } \mathsf{P}_\sigma) \\
&\equiv \mathsf{proj}_1 \, \Big(\mathsf{q}_A \, \big(\mathsf{TM}_\sigma \, (\mathsf{TM}_\Gamma \, \beta \, \rho), \mathsf{TM}_{A[\sigma]} \, \beta \, s, \mathsf{P}_\sigma \, (\mathsf{TM}_\sigma \, \rho, \mathsf{P}_\Gamma \, \beta \, \alpha), \mathsf{P}_A \, \beta \, a\big)\Big) \\
&= \mathsf{proj}_1 \, \big(\mathsf{q}_{A[\sigma]} \, (\mathsf{TM}_\Gamma \, \beta \, \rho, \mathsf{TM}_{A[\sigma]} \, \beta \, s, \mathsf{P}_\Gamma \, \beta \, \alpha, \mathsf{P}_{A[\sigma]} \, \beta \, a)\big)
\end{aligned}
$$

We need to verify that the two equalities $[][]$ and $[\mathsf{id}]$ hold in the interpretation. It is enough to check that the function parts of the natural transformations are equal.

The rule [][] is validated by associativity of substitution composition.

$$\begin{aligned}
&\mathsf{u}_{A[\sigma][\nu]}\,(\rho, n, \alpha)\\
=\,&\mathsf{u}_A\,\big(\mathsf{TM}_\sigma\,(\mathsf{TM}_\nu\,\rho), n, \mathsf{P}_\sigma\,(\mathsf{TM}_\nu\,\rho, \mathsf{P}_\nu\,(\rho, \alpha))\big)\\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad(\circ\circ)\\
\equiv\,&\mathsf{u}_A\,\big(\mathsf{TM}_{\sigma\circ\nu}\,\rho, n, \mathsf{P}_\sigma\,(\mathsf{TM}_\nu\,\rho, \mathsf{P}_\nu\,(\rho, \alpha))\big)\\
=\,&\mathsf{u}_{A[\sigma\circ\nu]}\,(\rho, n, \alpha)
\end{aligned}$$

$$\begin{aligned}
&\mathsf{q}_{A[\sigma][\nu]}\,(\rho, s, \alpha, a)\\
=\,&\mathsf{q}_A\,\big(\mathsf{TM}_\sigma\,(\mathsf{TM}_\nu\,\rho), s, \mathsf{P}_\sigma\,(\mathsf{TM}_\nu\,\rho, \mathsf{P}_\nu\,(\rho, \alpha)), a\big)\\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad(\circ\circ)\\
\equiv\,&\mathsf{q}_A\,(\mathsf{TM}_{\sigma\circ\nu}\,\rho, s, \mathsf{P}_\sigma\,(\mathsf{TM}_\nu\,\rho, \mathsf{P}_\nu\,(\rho, \alpha)), a)\\
=\,&\mathsf{q}_{A[\sigma\circ\nu]}\,(\rho, s, \alpha, a)
\end{aligned}$$

The rule [id] is validated by the left identity law for substitutions.

$$\begin{aligned}
&\mathsf{u}_{A[\mathsf{id}]}\,(\rho, n, \alpha)\\
=\,&\mathsf{u}_A\,(\mathsf{TM}_{\mathsf{id}}\,\rho, n, \alpha)\\
&\qquad\qquad\qquad(\mathsf{id}\circ)\\
\equiv\,&\mathsf{u}_A\,(\rho, n, \alpha)
\end{aligned}$$

$$\begin{aligned}
&\mathsf{q}_{A[\mathsf{id}]}\,(\rho, s, \alpha, a)\\
=\,&\mathsf{q}_A\,(\mathsf{TM}_{\mathsf{id}}\,\rho, s, \alpha, a)\\
&\qquad\qquad\qquad(\mathsf{id}\circ)\\
\equiv\,&\mathsf{q}_A\,(\rho, s, \alpha, a)
\end{aligned}$$

5.9.3 Methods for the base type and base family

Quote at the base type and base family are trivial, since in these cases the logical predicate says exactly the same as $\mathsf{NF}^{\equiv}$. For unquote, we just return the neutral term embedded into normal forms and the proof of equality is reflexivity.

The definitions at the base type are given as follows.

$$\begin{aligned}
&\mathsf{u}_{\mathsf{U}}\left((\rho : \mathsf{TM}_\Gamma\,\Psi), (n : \mathsf{Ne}\,\Psi\,\mathsf{U}[\rho]), \alpha\right) && : \Sigma(n' : \mathsf{NF}_{\mathsf{U}}\,\mathsf{id}).\ulcorner n \urcorner \equiv \ulcorner n' \urcorner \\
& && := \mathsf{neuU}\,({}_{\mathsf{U}[]*}n), \mathsf{refl} \\
&\mathsf{q}_{\mathsf{U}}\left(\rho, t, \alpha, \left(a : \mathsf{NF}^{\equiv}{}_{\mathsf{U}}\,(\mathsf{id}, t)\right)\right) && : \mathsf{NF}^{\equiv}{}_{\mathsf{U}}\,(\rho, t) := {}_{\mathsf{U}[]*}a
\end{aligned}$$

Naturality is trivial. It is enough to check the equality of the first components as the second components are equalities.

$$\begin{aligned}
&\mathsf{proj}_1\left(\mathsf{P}_{\mathsf{U}}\,\beta\,(\mathsf{u}_{\mathsf{U}}\,(\rho, n, \alpha))\right) = n[\beta] = \mathsf{proj}_1\left(\mathsf{u}_{\mathsf{U}}\,(\mathsf{TM}_\Gamma\,\beta\,\rho, \mathsf{NE}_{\mathsf{U}}\,\beta\,n, \mathsf{P}_\Gamma\,\beta\,\alpha)\right) \\
&\mathsf{NF}_{\mathsf{U}}\,\beta\left(\mathsf{proj}_1\,(\mathsf{q}_{\mathsf{U}}\,(\rho, t, \alpha, a))\right) = (\mathsf{proj}_1\,a)[\beta] \\
=\,&\mathsf{proj}_1\left(\mathsf{q}_{\mathsf{U}}\,(\mathsf{TM}_\Gamma\,\beta\,\rho, \mathsf{TM}_{\mathsf{U}}\,\beta\,t, \mathsf{P}_\Gamma\,\beta\,\alpha, \mathsf{P}_{\mathsf{U}}\,\beta\,a)\right)
\end{aligned}$$

Unqoute and quote at the base family are defined as follows.

$$\begin{aligned}
&\mathsf{u}_{\mathsf{El}\,\hat{A}}\left((\rho : \mathsf{TM}_\Gamma\,\Psi), (n : \mathsf{Ne}\,\Psi\,(\mathsf{El}\,\hat{A}[\rho])), \alpha\right) && : \Sigma(n' : \mathsf{NF}_{\mathsf{El}\,\hat{A}}\,\mathsf{id}).\ulcorner n \urcorner \equiv \ulcorner n' \urcorner \\
& && := \mathsf{neuEl}\,({}_{\mathsf{El}[]*}n), \mathsf{refl} \\
&\mathsf{q}_{\mathsf{El}\,\hat{A}}\left(\rho, t, \alpha, \left(a : \mathsf{NF}^{\equiv}{}_{\mathsf{El}\,\hat{A}}\,(\mathsf{id}, t)\right)\right) && : \mathsf{NF}^{\equiv}{}_{\mathsf{El}\,\hat{A}}\,(\rho, t) \\
& && := {}_{\mathsf{El}[]*}a
\end{aligned}$$

Naturality is trivial, the proof is the same as in the case of U.

As a last step in defining the methods for the base type and family, we verify the semantic counterparts of the laws $\mathsf{U}[]$ and $\mathsf{El}[]$.

$$\begin{aligned}
&\mathsf{u}_{\mathsf{U}[\sigma]}\,(\rho, n, \alpha) = \mathsf{u}_{\mathsf{U}}\left(\mathsf{TM}_\sigma\,\rho, n, \mathsf{P}_\sigma\,(\rho, \alpha)\right) = (n, \mathsf{refl}) = \mathsf{u}_{\mathsf{U}}\,(\rho, n, \alpha) \\
&\mathsf{q}_{\mathsf{U}[\sigma]}\,(\rho, t, \alpha, a) = \mathsf{q}_{\mathsf{U}}\left(\mathsf{TM}_\sigma\,\rho, t, \mathsf{P}_\sigma\,(\rho, \alpha), a\right) = a = \mathsf{u}_{\mathsf{U}}\,(\rho, t, \alpha, a) \\
&\mathsf{u}_{(\mathsf{El}\,\hat{A})[\sigma]}\,(\rho, n, \alpha) = \mathsf{u}_{\mathsf{El}\,\hat{A}}\left(\mathsf{TM}_\sigma\,\rho, n, \mathsf{P}_\sigma\,(\rho, \alpha)\right) = (n, \mathsf{refl}) \\
=\,&\mathsf{u}_{\mathsf{El}\,(\hat{A}[\sigma])}\,(\rho, n, \alpha) \\
&\mathsf{q}_{(\mathsf{El}\,\hat{A})[\sigma]}\,(\rho, t, \alpha, a) = \mathsf{q}_{\mathsf{El}\,\hat{A}}\left(\mathsf{TM}_\sigma\,\rho, t, \mathsf{P}_\sigma\,(\rho, \alpha), a\right) = a \\
=\,&\mathsf{u}_{\mathsf{El}\,(\hat{A}[\sigma])}\,(\rho, t, \alpha, a)
\end{aligned}$$

5.9.4 Methods for the function space

Unquoting a function says that a neutral function n maps inputs for which the predicate holds into outputs for which the predicate holds. We show this by first quoting the input thus getting a normal form, and then we can apply the normal form (denoted m) to the neutral function obtaining a neutral term at the codomain type. After unquoting this result we obtain the witness of the predicate at the result type and this is what we wanted. We write down the expanded types.

$$\begin{aligned}
&\mathsf{u}_{\Pi\,A\,B}\,\big((\rho : \mathsf{TM}_\Gamma\,\Psi), (n : \mathsf{NE}_{\Pi\,A\,B}\,\rho), \alpha\big)\\
&\quad : \Sigma\Big(\mathsf{map} : \big(\beta : \mathsf{REN}(\Omega,\Psi)\big)\big(u : \mathsf{TM}_A\,(\mathsf{TM}_\Gamma\,\beta\,\rho)\big)\\
&\qquad\qquad\big(v : \mathsf{P}_{A\,\Omega}\,(\mathsf{TM}_\Gamma\,\beta\,\rho, u, \mathsf{P}_\Gamma\,\beta\,\alpha)\big)\\
&\qquad\qquad \to \mathsf{P}_{B\,\Omega}\,\big((\mathsf{TM}_\Gamma\,\beta\,\rho, u), (\mathsf{TM}_{\Pi\,A\,B}\,\beta\,\ulcorner n\urcorner)\$u, (\mathsf{P}_\Gamma\,\beta\,\alpha, v)\big)\Big)\\
&\qquad .\forall\beta, u, v, \gamma.\mathsf{P}_B\,\gamma\,(\mathsf{map}\,\beta\,u\,v) \equiv \mathsf{map}\,(\beta\circ\gamma)\,(\mathsf{TM}_A\,\gamma\,u)\,(\mathsf{P}_A\,\gamma\,v)\\
&\quad := \lambda\beta\,u\,v.\mathsf{let}\,(m,p) := \mathsf{q}_A\,(\mathsf{TM}_\Gamma\,\beta\,\rho, u, \mathsf{P}_\Gamma\,\beta\,\alpha, v)\\
&\qquad\qquad\qquad\qquad\qquad : \Sigma\big(m : \mathsf{NF}_A\,(\mathsf{TM}_\Gamma\,\beta\,\rho)\big).u \equiv \ulcorner m\urcorner\\
&\qquad\qquad\mathsf{in}\;\;\mathsf{u}_B\,\big((\mathsf{TM}_\Gamma\,\beta\,\rho, u), ({}_{p*}\mathsf{app}\,(\mathsf{NE}_{\Pi\,A\,B}\,\beta\,n)\,m), (\mathsf{P}_\Gamma\,\beta\,\alpha, v)\big)\\
&\qquad , \lambda\beta\,u\,v\,\gamma.r
\end{aligned}$$

The naturality part of the function is given by the following equational reasoning denoted r. It combines the naturality of quote and unquote for the smaller types A and B.

$$\begin{aligned}
r :=& \mathsf{P}_B\,\gamma\,\Big(\mathsf{u}_B\,\big((\mathsf{TM}_\Gamma\,\beta\,\rho, u)\\
&\qquad\qquad , \mathsf{app}\,(\mathsf{NE}_{\Pi\,A\,B}\,\beta\,n)\,(\mathsf{proj}_1\,(\mathsf{q}_A\,(\mathsf{TM}_\Gamma\,\beta\,\rho, u, \mathsf{P}_\Gamma\,\beta\,\alpha, v)))\\
&\qquad\qquad , (\mathsf{P}_\Gamma\,\beta\,\alpha, v)\big)\Big)\\
&&\text{(naturality of } \mathsf{u}_B)\\
\equiv\;\;& \mathsf{u}_B\,\Big(\mathsf{TM}_{\Gamma,A}\,\gamma\,(\mathsf{TM}_\Gamma\,\beta\,\rho, u)\\
&\quad , \mathsf{NE}_B\,\gamma\,\big(\mathsf{app}\,(\mathsf{NE}_{\Pi\,A\,B}\,\beta\,n)\,(\mathsf{proj}_1\,(\mathsf{q}_A\,(\mathsf{TM}_\Gamma\,\beta\,\rho, u, \mathsf{P}_\Gamma\,\beta\,\alpha, v)))\big)\\
&\quad , \mathsf{P}_{\Gamma,A}\,\gamma\,\big(\mathsf{P}_\Gamma\,\beta\,\alpha, v\big)\Big)\\
=\;\;& \mathsf{u}_B\,\Big(\mathsf{TM}_{\Gamma,A}\,\gamma\,(\mathsf{TM}_\Gamma\,\beta\,\rho, u)\\
&\quad , \mathsf{app}\,\big(\mathsf{NE}_{\Pi\,A\,B}\,\gamma\,(\mathsf{NE}_{\Pi\,A\,B}\,\beta\,n)\big)
\end{aligned}$$

$$
\begin{aligned}
&\qquad \big(\mathsf{NF}_A\,\gamma\,(\mathsf{proj}_1\,(\mathsf{q}_A\,(\mathsf{TM}_\Gamma\,\beta\,\rho, u, \mathsf{P}_\Gamma\,\beta\,\alpha, v)))\big)\\
&\quad ,\mathsf{P}_{\Gamma,A}\,\gamma\,\big(\mathsf{P}_\Gamma\,\beta\,\alpha, v)\Big)\\
&\hfill (,\circ \text{ and naturality of } \mathsf{q}_A)\\
\equiv\;& \mathsf{u}_B\,\Big(\big(\mathsf{TM}_\Gamma\,\gamma\,(\mathsf{TM}_\Gamma\,\beta\,\rho), \mathsf{TM}_A\,\gamma\,u\big)\\
&\quad ,\mathsf{app}\,\big(\mathsf{NE}_{\Pi\,A\,B}\,\gamma\,(\mathsf{NE}_{\Pi\,A\,B}\,\beta\,n)\big)\\
&\qquad \big(\mathsf{proj}_1\,(\mathsf{q}_A\,(\mathsf{TM}_\Gamma\,\gamma\,(\mathsf{TM}_\Gamma\,\beta\,\rho), \mathsf{TM}_A\,\gamma\,u, \mathsf{P}_\Gamma\,\gamma\,(\mathsf{P}_\Gamma\,\beta\,\alpha), \mathsf{P}_A\,\gamma\,v))\big)\\
&\quad ,\big(\mathsf{P}_\Gamma\,\gamma\,(\mathsf{P}_\Gamma\,\beta\,\alpha), \mathsf{P}_A\,\gamma\,v\big)\Big)\\
&\hfill (\text{functor laws for } \mathsf{TM}_\Gamma,\ \mathsf{NE}_{\Pi\,A\,B} \text{ and } \mathsf{P}_\Gamma)\\
\equiv\;& \mathsf{u}_B\,\Big(\big(\mathsf{TM}_\Gamma\,(\beta\circ\gamma)\,\rho, \mathsf{TM}_A\,\gamma\,u\big)\\
&\quad ,\mathsf{app}\,\big(\mathsf{NE}_{\Pi\,A\,B}\,(\beta\circ\gamma)\,n\big)\\
&\qquad \big(\mathsf{proj}_1\,(\mathsf{q}_A\,(\mathsf{TM}_\Gamma\,(\beta\circ\gamma)\,\rho, \mathsf{TM}_A\,\gamma\,u, \mathsf{P}_\Gamma\,(\beta\circ\gamma)\,\alpha, \mathsf{P}_A\,\gamma\,v))\big)\\
&\quad ,\big(\mathsf{P}_\Gamma\,(\beta\circ\gamma)\,\alpha, \mathsf{P}_A\,\gamma\,v\big)\Big)
\end{aligned}
$$

Quoting a function has as input a term t which preserves the predicate and we would like to get a normal form t' which is equal to t. First we unquote the last variable in the extended context Γ, A thereby getting a witness a of the predicate at A. Then we quote $\mathsf{app}\,t$ at B supplying a as a witness that the predicate holds at this extended context. f tells us that the predicate holds for the result.

$$
\begin{aligned}
&\mathsf{q}_{\Pi\,A\,B}\,(\rho, t, \alpha, f) : \Sigma(t' : \mathsf{NF}_{\Pi\,A\,B}\,\rho).t \equiv \ulcorner t' \urcorner\\
&:= \mathsf{let}\,a \qquad := \mathsf{u}_A\,(\mathsf{TM}_\Gamma\,\mathsf{wk}\,\rho, \mathsf{var}\,\mathsf{vz}, \mathsf{P}_\Gamma\,\mathsf{wk}\,\alpha)\\
&\qquad\quad (t', p) := \mathsf{q}_B\,\big(\rho\uparrow, \mathsf{app}\,t, (\mathsf{P}_\Gamma\,\mathsf{wk}\,\alpha, a), \mathsf{map}\,f\,\mathsf{wk}\,\mathsf{vz}\,a\big)\\
&\qquad \mathsf{in}\ \ (\mathsf{lam}\,t', \Pi\eta^{-1} \centerdot \mathsf{ap}\,\mathsf{lam}\,p)
\end{aligned}
$$

p says that $\mathsf{app}\,t \equiv \ulcorner t' \urcorner$, hence we can construct the equality that we need by first using $\Pi\eta^{-1}$ i.e. $t \equiv \mathsf{lam}\,(\mathsf{app}\,t)$ and then $\mathsf{ap}\,\mathsf{lam}\,p$ which says $\mathsf{lam}\,(\mathsf{app}\,t) \equiv \mathsf{lam}\,\ulcorner t' \urcorner = \ulcorner \mathsf{lam}\,t' \urcorner$.

The definition of quote for the function space is the place which forces us to define unquote as we need a way to show that variables witness the predicate, and this is given by unquote. Also, this is where we need the Kripke function space: we need to use the semantic function f in a future world (the extended context) and this can be achieved using wk which takes us to this future world.

The choice of using REN as the base category becomes clear here: we need a wide subcategory of the category of contexts and substitutions which at least includes wk and REN is a convenient choice (the category of weakenings could be another choice).

Now we check that the above definitions are natural. We don't check equality of equalities. Naturality of unquote at Π follows from the functor laws for TM_Γ, $\mathsf{NE}_{\Pi\,A\,B}$ and P_Γ.

$$
\begin{aligned}
& \mathsf{P}_{\Pi\,A\,B}\,\beta'\,\Big(\mathsf{map}\,\big(\mathsf{u}_{\Pi\,A\,B}\,(\rho, n, \alpha)\big)\Big) \\
= {} & \lambda\beta\,u\,v.\mathsf{u}_B\,\Big((\mathsf{TM}_\Gamma\,(\beta' \circ \beta)\,\rho, u) \\
& \qquad\qquad , \mathsf{app}\,\big(\mathsf{NE}_{\Pi\,A\,B}\,(\beta' \circ \beta)\,n\big) \\
& \qquad\qquad\qquad \big(\mathsf{proj}_1\,(\mathsf{q}_A\,(\mathsf{TM}_\Gamma\,(\beta' \circ \beta)\,\rho, u, \mathsf{P}_\Gamma\,(\beta' \circ \beta)\,\alpha, v))\big) \\
& \qquad\qquad , \big(\mathsf{P}_\Gamma\,(\beta' \circ \beta)\,\alpha, v\big)\Big) \\
& \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad (\text{functoriality of } \mathsf{TM}_\Gamma,\ \mathsf{NE}_{\Pi\,A\,B} \text{ and } \mathsf{P}_\Gamma) \\
\equiv {} & \lambda\beta\,u\,v.\mathsf{u}_B\,\Big((\mathsf{TM}_\Gamma\,\beta\,(\mathsf{TM}_\Gamma\,\beta'\,\rho), u) \\
& \qquad\qquad , \mathsf{app}\,\big(\mathsf{NE}_{\Pi\,A\,B}\,\beta\,(\mathsf{NE}_{\Pi\,A\,B}\,\beta'\,n)\big) \\
& \qquad\qquad\qquad \big(\mathsf{proj}_1\,(\mathsf{q}_A\,(\mathsf{TM}_\Gamma\,\beta\,(\mathsf{TM}_\Gamma\,\beta'\,\rho), u, \mathsf{P}_\Gamma\,\beta\,(\mathsf{P}_\Gamma\,\beta'\,\alpha), v))\big) \\
& \qquad\qquad , \big(\mathsf{P}_\Gamma\,\beta\,(\mathsf{P}_\Gamma\,\beta'\,\alpha), v\big)\Big) \\
= {} & \mathsf{map}\,\big(\mathsf{u}_{\Pi\,A\,B}\,(\mathsf{TM}_\Gamma\,\beta'\,\rho, \mathsf{NE}_{\Pi\,A\,B}\,\beta'\,n, \mathsf{P}_\Gamma\,\beta'\,\alpha)\big)
\end{aligned}
$$

Naturality of quote for $\Pi\,A\,B$ depends on naturality of quote for B and unquote for A and also the naturality of f.

$$
\begin{aligned}
& \mathsf{NF}_{\Pi\,A\,B}\,\beta\,\Big(\mathsf{proj}_1\,\big(\mathsf{q}_{\Pi\,A\,B}\,(\rho, t, \alpha, f)\big)\Big) \\
= {} & \mathsf{NF}_{\Pi\,A\,B}\,\beta\,\Bigg(\mathsf{lam}\,\Bigg(\mathsf{let}\,a := \mathsf{u}_A\,(\mathsf{TM}_\Gamma\,\mathsf{wk}\,\rho, \mathsf{var}\,\mathsf{vz}, \mathsf{P}_\Gamma\,\mathsf{wk}\,\alpha) \\
& \qquad\qquad\qquad \mathsf{in}\ \ \mathsf{proj}_1\,\big(\mathsf{q}_B\,(\rho\uparrow, \mathsf{app}\,t, (\mathsf{P}_\Gamma\,\mathsf{wk}\,\alpha, a), \mathsf{map}\,f\,\mathsf{wk}\,\mathsf{vz}\,a)\big)\Bigg)\Bigg) \\
= {} & \mathsf{lam}\,\Big(\mathsf{let}\,a := \mathsf{u}_A\,(\mathsf{TM}_\Gamma\,\mathsf{wk}\,\rho, \mathsf{var}\,\mathsf{vz}, \mathsf{P}_\Gamma\,\mathsf{wk}\,\alpha) \\
& \qquad\quad \mathsf{in}\ \ \mathsf{NF}_B\,(\beta\uparrow)\,\big(\mathsf{proj}_1\,\big(\mathsf{q}_B\,(\rho\uparrow, \mathsf{app}\,t, (\mathsf{P}_\Gamma\,\mathsf{wk}\,\alpha, a), \mathsf{map}\,f\,\mathsf{wk}\,\mathsf{vz}\,a)\big)\big)\Big) \\
& \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad (\text{naturality of } \mathsf{q}_B) \\
\equiv {} & \mathsf{lam}\,\Big(\mathsf{let}\,a := \mathsf{u}_A\,(\mathsf{TM}_\Gamma\,\mathsf{wk}\,\rho, \mathsf{var}\,\mathsf{vz}, \mathsf{P}_\Gamma\,\mathsf{wk}\,\alpha)
\end{aligned}
$$

$$\begin{aligned}
&\qquad \mathsf{in}\ \ \mathsf{proj}_1\,\big(\mathsf{q}_B\,(\mathsf{TM}_{\Gamma,A}\,(\beta\uparrow)\,(\rho\uparrow), \mathsf{TM}_B\,(\beta\uparrow)\,(\mathsf{app}\,t)\\
&\qquad\qquad\qquad , \mathsf{P}_{\Gamma,A}\,(\beta\uparrow)\,(\mathsf{P}_\Gamma\,\mathsf{wk}\,\alpha, a), \mathsf{P}_B\,(\beta\uparrow)\,(\mathsf{map}\,f\,\mathsf{wk}\,\mathsf{vz}\,a)))\Big)\\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad \text{(substitution calculus, } \mathsf{app}[]\text{, } \mathsf{nat}\,f\text{)}\\
\equiv\ &\mathsf{lam}\,\Big(\mathsf{let}\,a := \mathsf{P}_A\,(\beta\uparrow)\,\big(\mathsf{u}_A\,(\mathsf{TM}_\Gamma\,\mathsf{wk}\,\rho, \mathsf{var}\,\mathsf{vz}, \mathsf{P}_\Gamma\,\mathsf{wk}\,\alpha)\big)\\
&\qquad \mathsf{in}\ \ \mathsf{proj}_1\,\big(\mathsf{q}_B\,((\mathsf{TM}_\Gamma\,\beta\,\rho)\uparrow, \mathsf{app}\,(\mathsf{TM}_{\Pi\,A\,B}\,\beta\,t)\\
&\qquad\qquad\qquad , (\mathsf{P}_\Gamma\,\mathsf{wk}\,(\mathsf{P}_\Gamma\,\beta\,\alpha), a), \mathsf{map}\,f\,(\beta\circ\mathsf{wk})\,\mathsf{vz}\,a)\big)\Big)\\
&\qquad \text{(naturality of } \mathsf{u}_A\text{, substitution calculus and definition of the action on}\\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \text{morphisms for } \mathsf{P}_{\Pi\,A\,B}\text{)}\\
\equiv\ &\mathsf{lam}\,\Big(\mathsf{let}\,a := \mathsf{u}_A\,\big(\mathsf{TM}_\Gamma\,\mathsf{wk}\,(\mathsf{TM}_\Gamma\,\beta\,\rho), \mathsf{var}\,\mathsf{vz}, \mathsf{P}_\Gamma\,\mathsf{wk}\,(\mathsf{P}_\Gamma\,\beta\,\alpha)\big)\\
&\qquad \mathsf{in}\ \ \mathsf{proj}_1\,\big(\mathsf{q}_B\,((\mathsf{TM}_\Gamma\,\beta\,\rho)\uparrow, \mathsf{app}\,(\mathsf{TM}_{\Pi\,A\,B}\,\beta\,t)\\
&\qquad\qquad\qquad , (\mathsf{P}_\Gamma\,\mathsf{wk}\,(\mathsf{P}_\Gamma\,\beta\,\alpha), a), \mathsf{map}\,(\mathsf{P}_{\Pi\,A\,B}\,\beta\,f)\,\mathsf{wk}\,\mathsf{vz}\,a)\big)\Big)\\
=\ &\mathsf{proj}_1\,\big(\mathsf{q}_{\Pi\,A\,B}\,(\mathsf{TM}_\Gamma\,\beta\,\rho, \mathsf{TM}_{\Pi\,A\,B}\,\beta\,t, \mathsf{P}_\Gamma\,\beta\,\alpha, \mathsf{P}_{\Pi\,A\,B}\,\beta\,f)\big)
\end{aligned}$$

The next step is verifying that quote and unquote preserve the equality $\Pi[]$. This is straightforward in both cases.

$$\begin{aligned}
&\mathsf{map}\,\big(\mathsf{u}_{(\Pi\,A\,B)[\sigma]}\,(\rho, n, \alpha)\big)\\
=\ &\mathsf{map}\,\Big(\mathsf{u}_{\Pi\,A\,B}\,\big(\mathsf{TM}_\sigma\,\rho, n, \mathsf{P}_\sigma\,(\rho,\alpha)\big)\Big)\\
=\ &\lambda\beta\,u\,v.\mathsf{u}_B\,\Big(\big(\mathsf{TM}_\Gamma\,\beta\,(\mathsf{TM}_\sigma\,\rho), u\big)\\
&\qquad\qquad , \mathsf{app}\,(\mathsf{NE}_{\Pi\,A\,B}\,\beta\,n)\\
&\qquad\qquad\qquad \big(\mathsf{proj}_1\,(\mathsf{q}_A\,(\mathsf{TM}_\Gamma\,\beta\,(\mathsf{TM}_\sigma\,\rho), u, \mathsf{P}_\Gamma\,\beta\,(\mathsf{P}_\sigma\,(\rho,\alpha)), v))\big)\\
&\qquad\qquad , \big(\mathsf{P}_\Gamma\,\beta\,(\mathsf{P}_\sigma\,(\rho,\alpha)), v\big)\Big)\\
&\qquad\qquad\qquad \text{(substitution calculus, } \mathsf{P}_\uparrow\text{, and naturality of } \mathsf{TM}_\sigma \text{ and } \mathsf{P}_\sigma\text{)}\\
\equiv\ &\lambda\beta\,u\,v.\mathsf{u}_B\,\Big(\mathsf{TM}_{\sigma\uparrow}\,(\mathsf{TM}_\Theta\,\beta\,\rho, u)\\
&\qquad\qquad , \mathsf{app}\,(\mathsf{NE}_{\Pi\,A\,B}\,\beta\,n)\\
&\qquad\qquad\qquad \big(\mathsf{proj}_1\,(\mathsf{q}_A\,(\mathsf{TM}_\sigma\,(\mathsf{TM}_\Theta\,\beta\,\rho), u, \mathsf{P}_\sigma\,(\mathsf{TM}_\Theta\,\beta\,\rho, \mathsf{P}_\Theta\,\beta\,\alpha), v))\big)\\
&\qquad\qquad , \mathsf{P}_{\sigma\uparrow}\,\big((\mathsf{TM}_\Theta\,\beta\,\rho, u), \mathsf{P}_\Theta\,\beta\,\alpha\big), v\Big)\\
=\ &\lambda\beta\,u\,v.\mathsf{u}_{B[\sigma\uparrow]}\,\Big(\big(\mathsf{TM}_\Theta\,\beta\,\rho, u\big)\\
&\qquad\qquad , \mathsf{app}\,(\mathsf{NE}_{\Pi\,A\,B}\,\beta\,n)\,\big(\mathsf{proj}_1\,(\mathsf{q}_{A[\sigma]}\,(\mathsf{TM}_\Theta\,\beta\,\rho, u, \mathsf{P}_\Theta\,\beta\,\alpha, v))\big)
\end{aligned}$$

$$
\begin{aligned}
&\qquad\qquad , \left(\mathsf{P}_\Theta\, \beta\, \alpha, v\right)\Big)\\
&= \mathsf{map}\, \left(\mathsf{u}_{\Pi\, A[\sigma]\, B[\sigma\uparrow]}\, (\rho, n, \alpha)\right)
\end{aligned}
$$

$$
\begin{aligned}
&\mathsf{proj}_1\, \left(\mathsf{q}_{(\Pi\, A\, B)[\sigma]}\, (\rho, t, \alpha, f)\right)\\
&= \mathsf{proj}_1\, \Big(\mathsf{q}_{\Pi\, A\, B}\, \left(\mathsf{TM}_\sigma\, \rho, t, \mathsf{P}_\sigma\, (\rho, \alpha), f\right)\Big)\\
&= \mathsf{lam}\Big(\mathsf{let}\, a := \mathsf{u}_A\, \left(\mathsf{TM}_\Gamma\, \mathsf{wk}\, (\mathsf{TM}_\sigma\, \rho), \mathsf{var}\, \mathsf{vz}, \mathsf{P}_\Gamma\, \mathsf{wk}\, (\mathsf{P}_\sigma\, (\rho, \alpha))\right)\\
&\qquad\quad \mathsf{in}\ \ \mathsf{proj}_1\, \left(\mathsf{q}_B\, ((\mathsf{TM}_\sigma\, \rho)\uparrow, \mathsf{app}\, t, (\mathsf{P}_\Gamma\, \mathsf{wk}\, (\mathsf{P}_\sigma\, (\rho, \alpha)), a), \mathsf{map}\, f\, \mathsf{wk}\, \mathsf{vz}\, a)\right)\Big)\\
&\qquad\qquad\qquad\qquad\qquad \text{(substitution calculus, } \mathsf{P}_\uparrow\text{, and naturality of } \mathsf{TM}_\sigma \text{ and } \mathsf{P}_\sigma\text{)}\\
&\equiv \mathsf{lam}\Big(\mathsf{let}\, a := \mathsf{u}_A\, \left(\mathsf{TM}_\sigma\, (\mathsf{TM}_\Theta\, \mathsf{wk}\, \rho), \mathsf{var}\, \mathsf{vz}, \mathsf{P}_\sigma\, (\mathsf{TM}_\Theta\, \mathsf{wk}\, \rho, \mathsf{P}_\Theta\, \mathsf{wk}\, \alpha)\right)\\
&\qquad\quad \mathsf{in}\ \ \mathsf{proj}_1\, \left(\mathsf{q}_B\, (\mathsf{TM}_{\sigma\uparrow}\, (\rho\uparrow), \mathsf{app}\, t, \mathsf{P}_{\sigma\uparrow}(\rho\uparrow, (\mathsf{P}_\Theta\, \mathsf{wk}\, \alpha, a)), \mathsf{map}\, f\, \mathsf{wk}\, \mathsf{vz}\, a)\right)\Big)\\
&= \mathsf{lam}\Big(\mathsf{let}\, a := \mathsf{u}_{A[\sigma]}\, (\mathsf{TM}_\Theta\, \mathsf{wk}\, \rho, \mathsf{var}\, \mathsf{vz}, \mathsf{P}_\Theta\, \mathsf{wk}\, \alpha)\\
&\qquad\quad \mathsf{in}\ \ \mathsf{proj}_1\, \left(\mathsf{q}_{B[\sigma\uparrow]}\, (\rho\uparrow, \mathsf{app}\, t, (\mathsf{P}_\Theta\, \mathsf{wk}\, \alpha, a), \mathsf{map}\, f\, \mathsf{wk}\, \mathsf{vz}\, a)\right)\Big)\\
&= \mathsf{proj}_1\, \left(\mathsf{q}_{\Pi\, A[\sigma]\, B[\sigma\uparrow]}\, (\rho, t, \alpha, f)\right)
\end{aligned}
$$

This concludes the definition of quote and unquote as the methods for substitutions and terms are trivial.

5.10 Normalisation

Using the definition of the logical predicate and quote we can define normalisation and show completeness as follows.

$$
\begin{aligned}
&\mathsf{norm}_A\ \ (t : \mathsf{Tm}\, \Gamma\, A) : \mathsf{Nf}\, \Gamma\, A && := \mathsf{proj}_1\, \Big(\mathsf{q}_A\, \left(\mathsf{id}, {}_{[\mathsf{id}]^{-1}*}t, \mathsf{u}_\Gamma\, \mathsf{id}, \mathsf{P}_t\, (\mathsf{id}, \mathsf{u}_\Gamma\, \mathsf{id})\right)\Big)\\
&\mathsf{compl}_A\, (t : \mathsf{Tm}\, \Gamma\, A) : t \equiv \ulcorner \mathsf{norm}_A\, t \urcorner && := \mathsf{proj}_2\, \Big(\mathsf{q}_A\, \left(\mathsf{id}, {}_{[\mathsf{id}]^{-1}*}t, \mathsf{u}_\Gamma\, \mathsf{id}, \mathsf{P}_t\, (\mathsf{id}, \mathsf{u}_\Gamma\, \mathsf{id})\right)\Big)
\end{aligned}
$$

We use quote to produce an element of $\mathsf{NF}^{\equiv}{}_A$: as arguments we supply the identity substitution, the term itself (which needs to be transported along $[\mathsf{id}]$ to get a type which is substituted by identity), the unquoted identity (neutral) substitution and the fundamental theorem of the logical predicate at the term t. This needs a starting point i.e. a substitution for which the logical relation holds, and again, this is given by the identity substitution and unquoting it.

We prove stability by mutual induction on neutral terms and normal forms.

$$\frac{n : \mathsf{Ne}\,\Gamma\,A}{\mathsf{stab}_n : \mathsf{P}_{\ulcorner n \urcorner}\,(\mathsf{id}, \mathsf{u}_\Gamma\,\mathsf{id}) \equiv \mathsf{u}_A\,(\mathsf{id}, n, \mathsf{u}_\Gamma\,\mathsf{id})} \qquad \frac{v : \mathsf{Nf}\,\Gamma\,A}{\mathsf{stab}_v : \mathsf{norm}_A\,\ulcorner v \urcorner \equiv v}$$

For neutral terms, we have the following cases.

$$\begin{aligned}
\mathsf{stab}_{\mathsf{var}\,\mathsf{vz}} \;:\;& \mathsf{P}_{\ulcorner \mathsf{var}\,\mathsf{vz} \urcorner}\,(\mathsf{id}, \mathsf{u}_{\Gamma,A}\,\mathsf{id}) \\
&= \mathsf{P}_{\mathsf{vz}}\,(\mathsf{id}, \mathsf{u}_{\Gamma,A}\,\mathsf{id}) \\
&= \mathsf{u}_A\,(\ulcorner \mathsf{wkV}\,\mathsf{id} \urcorner, \mathsf{var}\,\mathsf{vz}, \mathsf{u}_\Gamma\,\mathsf{id}) \\
&\equiv \mathsf{u}_A\,\big(\mathsf{TM}_{\mathsf{wk}}\,\mathsf{id}, \mathsf{var}\,\mathsf{vz}, \mathsf{P}_{\mathsf{wk}}\,(\mathsf{id}, \mathsf{u}_{\Gamma,A}\,\mathsf{id})\big) \\
&= \mathsf{u}_{A[\mathsf{wk}]}\,(\mathsf{id}, \mathsf{var}\,\mathsf{vz}, \mathsf{u}_{\Gamma,A}\,\mathsf{id}) \\
\mathsf{stab}_{\mathsf{var}\,(\mathsf{vs}\,x)} \;:\;& \mathsf{P}_{\ulcorner \mathsf{var}\,(\mathsf{vs}\,x) \urcorner}\,(\mathsf{id}, \mathsf{u}_{\Gamma,B}\,\mathsf{id}) \\
&= \mathsf{P}_{\ulcorner x \urcorner[\mathsf{wk}]}\,(\mathsf{id}, \mathsf{u}_{\Gamma,B}\,\mathsf{id}) \\
&= \mathsf{P}_{\ulcorner x \urcorner}\,\big(\mathsf{TM}_{\mathsf{wk}}\,\mathsf{id}, \mathsf{u}_\Gamma\,(\mathsf{wkV}\,\mathsf{id})\big) \\
&\equiv \mathsf{P}_{\ulcorner x \urcorner}\,\big(\mathsf{TM}_\Gamma\,\mathsf{wk}\,\mathsf{id}, \mathsf{u}_\Gamma\,(\mathsf{NE}_\Gamma\,\mathsf{wk}\,\mathsf{id})\big) \\
&\qquad\qquad\qquad\qquad\qquad\qquad \text{(naturality of } \mathsf{u}_\Gamma \text{ and } \mathsf{P}_{\ulcorner t \urcorner}\text{)} \\
&\equiv \mathsf{P}_A\,\mathsf{wk}\,\big(\mathsf{P}_{\ulcorner x \urcorner}\,(\mathsf{id}, \mathsf{u}_\Gamma\,\mathsf{id})\big) \\
&\qquad\qquad\qquad\qquad\qquad\qquad (\mathsf{stab}_{\mathsf{var}\,x}) \\
&\equiv \mathsf{P}_A\,\mathsf{wk}\,\big(\mathsf{u}_A\,(\mathsf{id}, \mathsf{var}\,x, \mathsf{u}_\Gamma\,\mathsf{id})\big) \\
&\qquad\qquad\qquad\qquad\qquad\qquad \text{(naturality of } \mathsf{u}_\Gamma \text{ and } \mathsf{u}_A\text{)} \\
&\equiv \mathsf{u}_A\,\big(\mathsf{TM}_\Gamma\,\mathsf{wk}\,\mathsf{id}, \mathsf{NE}_\Gamma\,\mathsf{wk}\,(\mathsf{var}\,x), \mathsf{u}_\Gamma\,(\mathsf{NE}_\Gamma\,\mathsf{wk}\,\mathsf{id})\big) \\
&\equiv \mathsf{u}_A\,\big(\mathsf{TM}_{\mathsf{wk}}\,\mathsf{id}, \mathsf{NE}_\Gamma\,\mathsf{wk}\,(\mathsf{var}\,x), \mathsf{u}_\Gamma\,(\mathsf{wkV}\,\mathsf{id})\big) \\
&= \mathsf{u}_{A[\mathsf{wk}]}\,\big(\mathsf{id}, \mathsf{var}\,(\mathsf{vs}\,x), \mathsf{u}_{\Gamma,B}\,\mathsf{id}\big) \\
\mathsf{stab}_{\mathsf{app}\,n\,v} \;:\;& \mathsf{P}_{\ulcorner \mathsf{app}\,n\,v \urcorner}\,(\mathsf{id}, \mathsf{u}_\Gamma\,\mathsf{id}) \\
&= \mathsf{P}_{(\mathsf{app}\,\ulcorner n \urcorner)[\langle \ulcorner v \urcorner \rangle]}\,(\mathsf{id}, \mathsf{u}_\Gamma\,\mathsf{id}) \\
&= \mathsf{P}_{\mathsf{app}\,\ulcorner n \urcorner}\,\big((\mathsf{id}, \ulcorner v \urcorner), (\mathsf{u}_\Gamma\,\mathsf{id}, \mathsf{P}_{\ulcorner v \urcorner}\,(\mathsf{id}, \mathsf{u}_\Gamma\,\mathsf{id}))\big) \\
&= \mathsf{map}\,\big(\mathsf{P}_{\ulcorner n \urcorner}\,(\mathsf{id}, \mathsf{u}_\Gamma\,\mathsf{id})\big)\,\mathsf{id}\,\ulcorner v \urcorner\,\big(\mathsf{P}_{\ulcorner v \urcorner}\,(\mathsf{id}, \mathsf{u}_\Gamma\,\mathsf{id})\big) \\
&\qquad\qquad\qquad\qquad\qquad\qquad (\mathsf{stab}_n) \\
&\equiv \mathsf{map}\,\big(\mathsf{u}_{\Pi\,A\,B}\,(\mathsf{id}, n, \mathsf{u}_\Gamma\,\mathsf{id})\big)\,\mathsf{id}\,\ulcorner v \urcorner\,\big(\mathsf{P}_{\ulcorner v \urcorner}\,(\mathsf{id}, \mathsf{u}_\Gamma\,\mathsf{id})\big) \\
&= \mathsf{u}_B\,\Big((\mathsf{id}, \ulcorner v \urcorner), \mathsf{app}\,n\,\big(\mathsf{proj}_1\,(\mathsf{q}_A\,(\mathsf{id}, \ulcorner v \urcorner, \mathsf{u}_\Gamma\,\mathsf{id}, \mathsf{P}_{\ulcorner v \urcorner}\,(\mathsf{id}, \mathsf{u}_\Gamma\,\mathsf{id})))\big) \\
&\qquad , \big(\mathsf{u}_\Gamma\,\mathsf{id}, \mathsf{P}_{\ulcorner v \urcorner}\,(\mathsf{id}, \mathsf{u}_\Gamma\,\mathsf{id})\big)\Big)
\end{aligned}$$

$$
\begin{aligned}
& && (\mathsf{stab}_v) \\
&\equiv \mathsf{u}_B \Big((\mathsf{id}, \ulcorner v \urcorner), \mathsf{app}\, n\, v, \big(\mathsf{u}_\Gamma\, \mathsf{id}, \mathsf{P}_{\ulcorner v \urcorner}\, (\mathsf{id}, \mathsf{u}_\Gamma\, \mathsf{id})\big)\Big) \\
&= \mathsf{u}_{B[\langle \ulcorner v \urcorner \rangle]}\, (\mathsf{id}, \mathsf{app}\, n\, v, \mathsf{u}_\Gamma\, \mathsf{id})
\end{aligned}
$$

In the case of the zero De Bruijn index, the proof is just reasoning in the substitution calculus, for the successor case we need to use naturality of unquote and stability at the structurally smaller case. In the case of application, we use stability at the neutral function and at the normal argument.

For normal forms, we proceed as follows.

$$
\begin{aligned}
\mathsf{stab}_{\mathsf{neuU}\, n} \;:\;& \mathsf{norm}_{\mathsf{U}}\, \ulcorner \mathsf{neuU}\, n \urcorner \\
&= \mathsf{proj}_1 \Big(\mathsf{q}_{\mathsf{U}}\, \big(\mathsf{id}, \ulcorner n \urcorner, \mathsf{u}_\Gamma\, \mathsf{id}, \mathsf{P}_{\ulcorner n \urcorner}\, (\mathsf{id}, \mathsf{u}_\Gamma\, \mathsf{id})\big)\Big) \\
&= \mathsf{proj}_1 \big(\mathsf{P}_{\ulcorner n \urcorner}\, (\mathsf{id}, \mathsf{u}_\Gamma\, \mathsf{id})\big) \\
& && (\mathsf{stab}_n) \\
&\equiv \mathsf{proj}_1 \big(\mathsf{u}_{\mathsf{U}}\, (\mathsf{id}, n, \mathsf{u}_\Gamma\, \mathsf{id})\big) \\
&= \mathsf{neuU}\, n \\
\mathsf{stab}_{\mathsf{neuEl}\, n} \;:\;& \mathsf{norm}_{\mathsf{El}\, \hat{A}}\, \ulcorner \mathsf{neuEl}\, n \urcorner \\
&= \mathsf{proj}_1 \big(\mathsf{P}_{\ulcorner n \urcorner}\, (\mathsf{id}, \mathsf{u}_\Gamma\, \mathsf{id})\big) \\
& && (\mathsf{stab}_n) \\
&\equiv \mathsf{proj}_1 \big(\mathsf{u}_{\mathsf{El}\, \hat{A}}\, (\mathsf{id}, n, \mathsf{u}_\Gamma\, \mathsf{id})\big) \\
&= \mathsf{neuEl}\, n \\
\mathsf{stab}_{\mathsf{lam}\, v} \;:\;& \mathsf{norm}_{\Pi\, A\, B}\, \ulcorner \mathsf{lam}\, v \urcorner \\
&= \mathsf{proj}_1 \Big(\mathsf{q}_{\Pi\, A\, B}\, \big(\mathsf{id}, \mathsf{lam}\, \ulcorner v \urcorner, \mathsf{u}_\Gamma\, \mathsf{id}, \mathsf{P}_{\mathsf{lam}\, \ulcorner v \urcorner}\, (\mathsf{id}, \mathsf{u}_\Gamma\, \mathsf{id})\big)\Big) \\
&\equiv \mathsf{lam}\, \Big(\mathsf{let}\, a := \mathsf{u}_A\, \big(\mathsf{wk}, \mathsf{var}\, \mathsf{vz}, \mathsf{P}_\Gamma\, \mathsf{wk}\, (\mathsf{u}_\Gamma\, \mathsf{id})\big) \\
&\qquad\quad \mathsf{in}\;\; \mathsf{proj}_1\, \big(\mathsf{q}_B\, \big(\mathsf{id}, \mathsf{app}\, (\mathsf{lam}\, \ulcorner v \urcorner), (\mathsf{P}_\Gamma\, \mathsf{wk}\, (\mathsf{u}_\Gamma\, \mathsf{id}), a) \\
&\qquad\qquad\qquad\qquad , \mathsf{map}\, (\mathsf{P}_{\mathsf{lam}\, \ulcorner v \urcorner}\, (\mathsf{id}, \mathsf{u}_\Gamma\, \mathsf{id}))\, \mathsf{wk}\, \mathsf{vz}\, a)\big)\Big) \\
& && (\Pi\beta, \text{ naturality of } \mathsf{u}_\Gamma) \\
&\equiv \mathsf{lam}\, \Big(\mathsf{let}\, a := \mathsf{u}_A\, (\mathsf{wk}, \mathsf{var}\, \mathsf{vz}, \mathsf{u}_\Gamma\, \mathsf{wk}) \\
&\qquad\quad \mathsf{in}\;\; \mathsf{proj}_1\, \big(\mathsf{q}_B\, \big(\mathsf{id}, \ulcorner v \urcorner, (\mathsf{u}_\Gamma\, \mathsf{wk}, a) \\
&\qquad\qquad\qquad\qquad , \mathsf{P}_{\ulcorner v \urcorner}\, ((\mathsf{wk}, \mathsf{vz}), (\mathsf{u}_\Gamma\, \mathsf{wk}, a))\big)\big)\Big)
\end{aligned}
$$

$$\begin{aligned}
&\qquad\qquad\qquad\text{(definition of } \mathsf{u}_{\Gamma,A} \text{ and substitution calculus)}\\
&\equiv \mathsf{lam}\left(\mathsf{proj}_1\left(\mathsf{u}_B\left(\mathsf{id}, \ulcorner v \urcorner, \mathsf{u}_{\Gamma,A}\,\mathsf{id}, \mathsf{P}_{\ulcorner v \urcorner}\left(\mathsf{id}, \mathsf{u}_{\Gamma,A}\,\mathsf{id}\right)\right)\right)\right)\\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad (\mathsf{stab}_v)\\
&\equiv \mathsf{lam}\, v
\end{aligned}$$

In the case of an element of the base type and base family we just use the induction hypotheses for the neutral term. In the case of abstraction, we use stability at the body of the lambda abstraction, but we need some work to transform the definition into a form where we can apply this induction hypothesis.

We fullfilled the requirements for the specification of normalisation by providing the following isomorphism between terms and normal forms.

$$\mathsf{compl}_A \circlearrowright \qquad \mathsf{norm}_A \downarrow \frac{\mathsf{Tm}\,\Gamma\,A}{\mathsf{Nf}\,\Gamma\,A} \uparrow \ulcorner - \urcorner \qquad \curvearrowright \mathsf{stab}$$

5.11 Consistency

Using normalisation, we can prove properties of our type theory by reasoning only about normal forms. E.g. if we prove a property $P : \mathsf{Tm}\,\Gamma\,A \to \mathsf{Set}$ for every normal form, i.e.

$$p : (v : \mathsf{Nf}\,\Gamma\,A) \to P\,\ulcorner v \urcorner,$$

we also get that it is true for every term:

$$\lambda t.{}_{(\mathsf{compl}_A\,t)^{-1}*}\big(p\,(\mathsf{norm}_A\,t)\big) : (t : \mathsf{Tm}\,\Gamma\,A) \to P\,t.$$

We use this method to prove consistency of our type theory, that is there is no element of the type $\mathsf{Tm}\,\cdot\,\mathsf{U}$. Another proof of consistency used the standard model (section 3.6).

We do mutual induction on variables, neutral terms and normal forms. We show that in the empty context there are no variables and neutral terms at all. For normal forms, we derive $\bot$ from witnesses that the context is empty and the type is U.

$$
\begin{aligned}
&\mathsf{cons}^{\mathsf{Var}} \;: \mathsf{Var}\,\Gamma\,A \to \cdot \equiv \Gamma \to \bot\\
&\mathsf{cons}^{\mathsf{Ne}} \;: \mathsf{Ne}\,\Gamma\,A \to \cdot \equiv \Gamma \to \bot\\
&\mathsf{cons}^{\mathsf{Nf}} \;: \mathsf{Nf}\,\Gamma\,A \to \cdot \equiv \Gamma \to A \equiv \mathsf{U} \to \bot\\
&\mathsf{cons}^{\mathsf{Var}} \big(\mathsf{vz} : \mathsf{Var}\,(\Gamma, A)\,(A[\mathsf{wk}])\big)\,p := \mathsf{disj}_{\cdot\,\Gamma,A}\,p\\
&\mathsf{cons}^{\mathsf{Var}} \big(\mathsf{vs}\,x : \mathsf{Var}\,(\Gamma, B)\,(A[\mathsf{wk}])\big)\,p := \mathsf{disj}_{\cdot\,\Gamma,B}\,p\\
&\mathsf{cons}^{\mathsf{Ne}}\;(\mathsf{var}\,x)\,p := \mathsf{cons}^{\mathsf{Var}}\,x\,p\\
&\mathsf{cons}^{\mathsf{Ne}}\;(\mathsf{app}\,n\,v)\,p := \mathsf{cons}^{\mathsf{Ne}}\,n\,p\\
&\mathsf{cons}^{\mathsf{Nf}}\;(\mathsf{neuU}\,n)\,p\,q := \mathsf{cons}^{\mathsf{Ne}}\,n\,p\\
&\mathsf{cons}^{\mathsf{Nf}}\;(\mathsf{neuEl}\,n)\,p\,q := \mathsf{cons}^{\mathsf{Ne}}\,n\,p\\
&\mathsf{cons}^{\mathsf{Nf}}\;(\mathsf{lam}\,v)\,p\,(q : \Pi\,A\,B \equiv \mathsf{U}) := \mathsf{disj}_{(\Pi\,A\,B)\,\mathsf{U}}\,q
\end{aligned}
$$

For variables, we use disjointness of $\cdot$ and Γ, A, for neutral terms, we use the induction hypotheses, and for normal abstractions we use disjointness of Π and U. The disjointness properties were proved in section 3.4.1.

Chapter 6

Conclusions and further work

In this thesis we observe that the syntax of type theory can be defined in a typed way as a quotient inductive inductive type (QIIT). We use four types to build up the syntax: the type of contexts, types, substitutions and terms. The conversion relations for each type are represented as the equalities (identity types) of these types, respectively. The equivalence relation, congruence and coercion rules in the conversion relation are given by those for the identity type, while the β and η conversion rules are added as equality constructors.

We show that reasoning in this representation of the syntax is possible by defining some functions (the polymorphic identity function, the predicate space etc.) and deriving some laws in the syntax such as the identity substitution law for terms.

To define functions out of the syntax one needs to provide the motives and methods for the eliminator of the syntax. Collecting the motives and methods of the non dependent eliminator together, we observe that they form a well-known notion of model of type theory called categories with families [48]. To define a dependent function from the syntax one needs to provide the motives and methods for the dependent eliminator which together form a "dependent model".

As a simple usage of the non dependent eliminator (recursor) we define models which justify disjointness of context constructors and type constructors. Using the dependent eliminator we can prove injectivity of type constructors.

We define the standard model where every object theoretic construct is mapped to its meta theoretic counterpart. For example, object theoretic function space is mapped to meta theoretic function space. In this interpretation, the semantic counterpart for all equality constructors are reflexivity which shows that the interpretation is strict.

Parametricity is a way to characterise abstraction properties. For example, it provides means to show that a function of type $\Pi(A : \mathsf{U}).A \to A$ is the identity function; the intuitive explanation is that the type A cannot be inspected, the function needs to work uniformly for all As. This is formalised by the idea that terms respect logical relations (and also unary logical relations, called logical predicates). To express the parametricity property of type theory, the language of type theory itself is enough: we can define the logical predicate interpretation as a syntactic translation. This is what we did in chapter 4 for our basic type theory. The logical predicate interpretation shows that our syntax is workable in practice. A possible application of this translation would be the automatic derivation of parametricity properties when reasoning in type theory. Another application is deriving the eliminator for a closed QIIT as logical predicates formalise the idea of a dependent family over (families of) types which is exactly what the motives of a dependent eliminator are. The methods for the eliminator and the equalities that the eliminator needs to satisfy can also be derived using logical relations.

Normalisation is a very important property of type theory. Its consequences are that the theory is consistent and that the conversion relation is decidable. These are important metatheoretic properties of a type theory. Furthermore, normalisation is the way of running programs written in type theory, hence it is essential for using type theory as a programming language. Normalisation is specified by a function which maps a term into a normal form which is equal to the original term. Normal forms are the subset of terms where no application is applied directly to a lambda. We prove normalisation using a model construction together with a quote function from the model back into the syntax. In fact, this function outputs normal forms instead of terms. By concatenating the eliminator and the quote function, we get normalisation. The (dependent) model we use is the presheaf logical predicate interpretation. This is similar to the logical predicate interpretation defined in chapter 4, however it targets the metatheory, not the syntax and is parameterised by a category. It can be seen as a dependent variant of the presheaf model of type theory [58].

Of course, this thesis is by far not the last word on internalising type theory in type theory.

As far as the definition of the syntax is concerned, we would like to extend it with Σ types, inductive types, universes and coinductive types. Furthermore, QIITs should be internalised to make the internalisation full, that is, to have every meta theoretic construct in our object theory (obviously, we would have fewer universes in the object theory than in the metatheory). To express QIITs (or the more general HIITs), we might need a cubical type theory [40].

We used extensional type theory as a convenient notation in this thesis instead of intensional type theory with axioms. This choice was justified by the fact that the latter theory is conservative over the former [56, 81]. We would like to investigate the option of using a cubical theory for formalisation which has a more concise notation than our formal metatheory.

As demonstrated by section 3.3, our formal syntax is still hard to use in practice. One could help this by defining an inductive recursive version of the syntax and maps back and forth where substitution is defined recursively. With this at hand we would not need to use explicit substitutions whenever we define terms and the notation would be closer to the informal syntax. Furthermore, we would also like to develop an untyped version of the syntax with preterms together with a typechecking algorithm which outputs terms in our well-typed syntax. The typechecking algorithm would need the proof of decidability of normal forms, which is also left as future work.

The difficulty in proving the decidability of equality of normal forms comes from normal forms being indexed by (non-normal) contexts and types. To check whether two application neutral terms

$$\mathsf{app}\,\big(n : \mathsf{Ne}\,\Gamma\,(\Pi\,A\,B)\big)\,(v : \mathsf{Nf}\,\Gamma\,A) : \mathsf{Ne}\,\Gamma\,(B[\langle v\rangle])$$

and

$$\mathsf{app}\,\big(n' : \mathsf{Ne}\,\Gamma\,(\Pi\,A'\,B')\big)\,(v' : \mathsf{Nf}\,\Gamma\,A') : \mathsf{Ne}\,\Gamma\,(B'[\langle v'\rangle])$$

are equal (knowing that $B[\langle v\rangle] \equiv B'[\langle v'\rangle]$), we first need to check whether the types A, A' and B, B' are equal, then we need to check equality of the neutral functions n and n' and finally the normal arguments v and v'. But types again include terms, so it seems that we can only decide equality of normal forms if they only contain normal types and normal contexts.

We used the internal logical predicate interpretation to derive the eliminator for a QIIT (section 4.3), however we did not show that this algorithm is correct. Correctness could be proven for certain well studied cases of inductive types such as W-types. We would need to show that for W-types our method gives the expected eliminator. We would also like to extend this method to open types (e.g. parameterised types such as that of lists or vectors) and coinductive types.

We defined a presheaf logical predicate interpretation by fixing the base category to the category of renamings and also fixing the Yoneda embedding TM over which the logical predicate has been defined. TM can be seen as a weak morphism of models from the syntax the presheaf model. This means that certain equations are only satisfied up to isomorphism (e.g. we don't have $\mathsf{TM}_{\Gamma,A} \equiv \Sigma\,\mathsf{TM}_\Gamma\,\mathsf{TM}_A$, but we do have $\mathsf{TM}_{\Gamma,A} \cong \Sigma\,\mathsf{TM}_\Gamma\,\mathsf{TM}_A$). We would like to investigate whether the presheaf logical predicate interpretation can be generalised to arbitrary weak presheaf models. More generally, we would like to understand the high level picture and the relationship between normalisation and categorical glueing as hinted in section 5.5.

The metatheory of quotient inductive types (not to mention quotient inductive inductive types) is not worked out yet. We believe that the setoid model [8] could be used to justify these. In addition, work on the more general higher inductive types would also validate these constructions.

Having an internal syntax of type theory opens up the exciting possibility of developing *template type theory*. We may define an interpretation of type theory by defining an algebra for the syntax and the interpretation of new constants in this algebra. We can then interpret code using these new principles by interpreting it in the given algebra. The new code can use all the conveniences of the host system such as implicit arguments and definable syntactic extensions. There are a number of exciting applications of this approach such as using presheaf models to justify guarded type theory [60]. Another example is to model the local state monad (Haskell's STM monad) in another presheaf category to be able to program with and reason about local state and other resources. In the extreme such a template type theory may allow us to start with a fairly small core because everything else can be programmed as templates. This may include the computational explanation of homotopy type theory by the cubical model — we may not have to build in univalence into our type theory.

Bibliography

[1] Martín Abadi, Luca Cardelli, Pierre-Louis Curien, and Jean-Jacques Lévy. Explicit substitutions. In *Proceedings of the 17th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '90, pages 31–46, New York, NY, USA, 1990. ACM.

[2] Michael Abbott, Thorsten Altenkirch, and Neil Ghani. Containers — constructing strictly positive types. *Theoretical Computer Science*, 342:3–27, September 2005. Applied Semantics: Selected Topics.

[3] Michael Gordon Abbott. *Categories of containers.* PhD thesis, University of Leicester, 2003.

[4] Andreas Abel. Towards normalization by evaluation for the $\beta\eta$-calculus of constructions. In *Functional and Logic Programming*, pages 224–239. Springer, 2010.

[5] Andreas Abel. Normalization by evaluation: dependent types and impredicativity, 2013. Habilitation thesis, Ludwig-Maximilians-Universität München.

[6] Andreas Abel, Thierry Coquand, and Peter Dybjer. Normalization by evaluation for Martin-Löf type theory with typed equality judgements. In *Logic in Computer Science, 2007. LICS 2007. 22nd Annual IEEE Symposium on*, pages 3–12. IEEE, 2007.

[7] Andreas Abel, Thierry Coquand, and Bassel Mannaa. On the decidability of conversion in type theory. In Silvia Ghilezan and Jelena Ivetić, editors, *22nd International Conference on Types for Proofs and Programs, TYPES 2016.* University of Novi Sad, 2016.

[8] Thorsten Altenkirch. Extensional equality in intensional type theory. In *14th Symposium on Logic in Computer Science*, pages 412 – 420, 1999.

[9] Thorsten Altenkirch. Normalisation by completeness. Talk given at the Workshop on Normalization by Evaluation in Los Angeles, August 2009.

[10] Thorsten Altenkirch and James Chapman. Big-step normalisation. *Journal of Functional Programming*, 19(3-4):311–333, 2009.

[11] Thorsten Altenkirch, Peter Dybjer, Martin Hofmann, and Phil Scott. Normalization by evaluation for typed lambda calculus with coproducts. In *16th Annual IEEE Symposium on Logic in Computer Science*, pages 303–310, 2001.

[12] Thorsten Altenkirch, Neil Ghani, Peter Hancock, Conor McBride, and Peter Morris. Indexed containers. *J. Funct. Program.*, 25, 2015.

[13] Thorsten Altenkirch, Martin Hofmann, and Thomas Streicher. Categorical reconstruction of a reduction free normalization proof. In David Pitt, David E. Rydeheard, and Peter Johnstone, editors, *Category Theory and Computer Science*, LNCS 953, pages 182–199, 1995.

[14] Thorsten Altenkirch, Martin Hofmann, and Thomas Streicher. Reduction-free normalisation for a polymorphic system. In *11th Annual IEEE Symposium on Logic in Computer Science*, pages 98–106, 1996.

[15] Thorsten Altenkirch, Martin Hofmann, and Thomas Streicher. Reduction-free normalisation for system F. Unpublished draft, 1997.

[16] Thorsten Altenkirch and Ambrus Kaposi. Normalisation by evaluation for dependent types. In Delia Kesner and Brigitte Pientka, editors, *1st International Conference on Formal Structures for Computation and Deduction (FSCD 2016)*, volume 52 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 6:1–6:16, Dagstuhl, Germany, 2016. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik.

[17] Thorsten Altenkirch and Ambrus Kaposi. Towards a cubical type theory without an interval. Unpublished draft, 2016.

[18] Thorsten Altenkirch and Ambrus Kaposi. Type theory in type theory using quotient inductive types. In Rastislav Bodik and Rupak Majumdar, editors, *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium*

on Principles of Programming Languages, POPL 2016, St. Petersburg, FL, USA, January 20 - 22, 2016, pages 18–29. ACM, 2016.

[19] Thorsten Altenkirch, Conor Mcbride, and Wouter Swierstra. Observational equality, now! In *PLPV '07: Proceedings of the 2007 workshop on Programming languages meets program verification*, pages 57–58. ACM, 2007.

[20] Thorsten Altenkirch, Peter Morris, Fredrik Nordvall Forsberg, and Anton Setzer. A categorical semantics for inductive-inductive definitions. In *CALCO*, pages 70–84, 2011.

[21] Robert Atkey, Neil Ghani, and Patricia Johann. A relationally parametric model of dependent type theory. In Suresh Jagannathan and Peter Sewell, editors, *The 41st Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '14, San Diego, CA, USA, January 20-21, 2014*, pages 503–516. ACM, 2014.

[22] Henning Basold, Herman Geuvers, and Niels van der Weide. Higher inductive types in programming. *Journal of Universal Computer Science*, 23(1):63–88, January 2017.

[23] Ulrich Berger and Helmut Schwichtenberg. An inverse of the evaluation functional for typed λ–calculus. In R. Vemuri, editor, *Proceedings of the Sixth Annual IEEE Symposium on Logic in Computer Science*, pages 203–211. IEEE Computer Society Press, Los Alamitos, 1991.

[24] Jean-Philippe Bernardy, Thierry Coquand, and Guilhem Moulin. A presheaf model of parametric type theory. *Electronic Notes in Theoretical Computer Science*, 319(C):67–82, December 2015.

[25] Jean-Philippe Bernardy, Patrik Jansson, and Ross Paterson. Proofs for free — parametricity for dependent types. *Journal of Functional Programming*, 22(02):107–152, 2012.

[26] Jean-Philippe Bernardy and Guilhem Moulin. A computational interpretation of parametricity. In *Proceedings of the 2012 27th Annual IEEE/ACM Symposium on Logic in Computer Science*, LICS '12, pages 135–144, Washington, DC, USA, 2012. IEEE Computer Society.

[27] Jean-Philippe Bernardy and Guilhem Moulin. Type-theory in color. In *Proceedings of the 18th ACM SIGPLAN International Conference on Functional Programming*, ICFP '13, pages 61–72, New York, NY, USA, 2013. ACM.

[28] Marc Bezem, Thierry Coquand, and Simon Huber. A model of type theory in cubical sets. In *19th International Conference on Types for Proofs and Programs (TYPES 2013)*, volume 26, pages 107–128, 2014.

[29] Richard Bird and Oege de Moor. *Algebra of Programming.* Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1997.

[30] Edwin Brady. Idris, a general-purpose dependently typed programming language: Design and implementation. *J. Funct. Program.*, 23(5):552–593, 2013.

[31] Matt Brown and Jens Palsberg. Self-representation in Girard's System U. *SIGPLAN Not.*, 50(1):471–484, January 2015.

[32] Matt Brown and Jens Palsberg. Breaking through the normalization barrier: A self-interpreter for F-omega. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '16, pages 5–17, New York, NY, USA, 2016. ACM.

[33] N. G. De Bruijn. Lambda calculus notation with nameless dummies, a tool for automatic formula manipulation, with application to the church-rosser theorem. *INDAG. MATH*, 34:381–392, 1972.

[34] Paolo Capriotti. Mutual and higher inductive types in homotopy type theory, 2014. Nottingham FP Lab Away Day.

[35] Paolo Capriotti and Ambrus Kaposi. Free applicative functors. In *Proceedings 5th Workshop on Mathematically Structured Functional Programming, MSFP 2014, Grenoble, France, 12 April 2014.*, pages 2–30, 2014.

[36] John Cartmell. Generalised algebraic theories and contextual categories. *Annals of Pure and Applied Logic*, 32:209–243, 1986.

[37] James Chapman. *Type checking and normalisation.* PhD thesis, University of Nottingham, 2008.

[38] James Chapman. Type theory should eat itself. *Electronic Notes in Theoretical Computer Science*, 228:21–36, January 2009.

[39] Jesper Cockx and Andreas Abel. Sprinkles of extensionality for your vanilla type theory. In Silvia Ghilezan and Jelena Ivetić, editors, *22nd International Conference on Types for Proofs and Programs, TYPES 2016*. University of Novi Sad, 2016.

[40] Cyril Cohen, Thierry Coquand, Simon Huber, and Anders Mörtberg. Cubical type theory: a constructive interpretation of the univalence axiom. December 2015.

[41] R. L. Constable, S. F. Allen, H. M. Bromley, W. R. Cleaveland, J. F. Cremer, R. W. Harper, D. J. Howe, T. B. Knoblock, N. P. Mendler, P. Panangaden, J. T. Sasaki, and S. F. Smith. *Implementing Mathematics with the Nuprl Proof Development System.* Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1986.

[42] Roy L. Crole. *Categories for types.* Cambridge mathematical textbooks. Cambridge University Press, Cambridge, New York, 1993.

[43] Nils Anders Danielsson. A formalisation of a dependently typed language as an inductive-recursive family. In Thorsten Altenkirch and Conor McBride, editors, *TYPES*, volume 4502 of *Lecture Notes in Computer Science*, pages 93–109. Springer, 2006.

[44] Olivier Danvy. *Type-directed partial evaluation.* Springer, 1999.

[45] Leonardo Mendonça de Moura, Soonho Kong, Jeremy Avigad, Floris van Doorn, and Jakob von Raumer. The Lean theorem prover (system description). In Amy P. Felty and Aart Middeldorp, editors, *Automated Deduction - CADE-25 - 25th International Conference on Automated Deduction, Berlin, Germany, August 1-7, 2015, Proceedings*, volume 9195 of *Lecture Notes in Computer Science*, pages 378–388. Springer, 2015.

[46] Dominique Devriese and Frank Piessens. Typed syntactic meta-programming. In *Proceedings of the 2013 ACM SIGPLAN International Conference on Functional Programming (ICFP 2013)*, pages 73–85. ACM, September 2013.

[47] Gabe Dijkstra. *Quotient inductive-inductive definitions.* PhD thesis, University of Nottingham, 2017.

[48] Peter Dybjer. Internal type theory. In *Lecture Notes in Computer Science*, pages 120–134. Springer, 1996.

[49] Peter Dybjer. Inductive families. *Formal Aspects of Computing*, 6:440–465, 1997.

[50] Peter Dybjer and Anton Setzer. A finite axiomatization of inductive-recursive definitions. In *Typed Lambda Calculi and Applications, volume 1581 of Lecture Notes in Computer Science*, pages 129–146. Springer, 1999.

[51] Harvey Friedman. *Equality between functionals*, pages 22–37. Springer Berlin Heidelberg, Berlin, Heidelberg, 1975.

[52] Neil Ghani, Patricia Johann, and Clément Fumex. Generic fibrational induction. *Logical Methods in Computer Science*, 8(2), 2012.

[53] Jean-Yves Girard, Paul Taylor, and Yves Lafont. *Proofs and types.* Cambridge University Press, New York, NY, USA, 1989.

[54] Robert Harper, Furio Honsell, and Gordon Plotkin. A framework for defining logics. *J. ACM*, 40(1):143–184, January 1993.

[55] Claudio Hermida, Uday S. Reddy, and Edmund P. Robinson. Logical relations and parametricity – a Reynolds programme for category theory and programming languages. *Electronic Notes in Theoretical Computer Science*, 303(0):149 – 180, 2014. Proceedings of the Workshop on Algebra, Coalgebra and Topology (WACT 2013).

[56] Martin Hofmann. Conservativity of equality reflection over intensional type theory. In *TYPES 95*, pages 153–164, 1995.

[57] Martin Hofmann. *Extensional concepts in intensional type theory.* Thesis. University of Edinburgh, Department of Computer Science, 1995.

[58] Martin Hofmann. Syntax and semantics of dependent types. In *Semantics and Logics of Computation*, pages 79–130. Cambridge University Press, 1997.

[59] Martin Hofmann and Thomas Streicher. The groupoid interpretation of type theory. In *In Venice Festschrift*, pages 83–111. Oxford University Press, 1996.

[60] Guilhem Jaber, Nicolas Tabareau, and Matthieu Sozeau. Extending type theory with forcing. In *Logic in Computer Science (LICS), 2012 27th Annual IEEE Symposium on*, pages 395–404. IEEE, 2012.

[61] Ambrus Kaposi. Agda formalisation for the thesis "Type theory in a type theory with quotient inductive types", 2016. Available online at the author's website.

[62] Chantal Keller and Marc Lasson. Parametricity in an impredicative sort. *CoRR*, abs/1209.6336, 2012.

[63] Nicolai Kraus. *Truncation Levels in Homotopy Type Theory.* PhD thesis, University of Nottingham, 2015.

[64] Nicolai Kraus. Constructions with non-recursive higher inductive types. In *Proceedings of the 31st Annual ACM/IEEE Symposium on Logic in Computer Science*, LICS '16, pages 595–604, New York, NY, USA, 2016. ACM.

[65] Nuo Li. *Quotient types in type theory.* Thesis. University of Nottingham, Department of Computer Science, 2015.

[66] Dan Licata. Running circles around (in) your proof assistant; or, quotients that compute, 2011. Blog post on the Homotopy Type Theory website.

[67] Peter LeFanu Lumsdaine and Mike Shulman. Semantics of higher inductive types, 2012. Note.

[68] Lorenzo Malatesta, Thorsten Altenkirch, Neil Ghani, Peter Hancock, and Conor McBride. Small induction recursion, indexed containers and dependent polynomials are equivalent, 2013. TLCA 2013.

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

[70] Per Martin-Löf. *Intuitionistic type theory*, volume 1 of *Studies in Proof Theory.* Bibliopolis, 1984.

[71] Per Martin-Löf. Constructive mathematics and computer programming. In *Proc. Of a Discussion Meeting of the Royal Society of London on Mathematical Logic and Programming Languages*, pages 167–184, Upper Saddle River, NJ, USA, 1985. Prentice-Hall, Inc.

[72] Per Martin-Löf. An intuitionistic theory of types. In *Twenty-five years of constructive type theory (Venice, 1995)*, volume 36 of *Oxford Logic Guides*, pages 127–172. Oxford Univ. Press, New York, 1998.

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

[74] Conor McBride. *Dependently typed functional programs and their proofs.* PhD thesis, University of Edinburgh, 1999.

[75] Conor McBride. Outrageous but meaningful coincidences: dependent type-safe syntax and evaluation. In Bruno C. d. S. Oliveira and Marcin Zalewski, editors, *Proceedings of the ACM SIGPLAN Workshop on Generic Programming*, pages 1–12. ACM, 2010.

[76] Conor McBride and James McKinna. Functional pearl: I am not a number — I am a free variable. In *Proceedings of the 2004 ACM SIGPLAN Workshop on Haskell*, Haskell '04, pages 1–9, New York, NY, USA, 2004. ACM.

[77] Guilhem Moulin. *Pure type systems with an internalized parametricity theorem.* Chalmers University of Technology, 2013. Licentiate thesis.

[78] Guilhem Moulin. *Internalizing Parametricity.* Chalmers University of Technology, 2016. PhD thesis.

[79] Fredrik Nordvall Forsberg. *Inductive-inductive definitions.* PhD thesis, Swansea University, 2013.

[80] Ulf Norell. *Towards a practical programming language based on dependent type theory.* PhD thesis, Chalmers University of Technology, 2007.

[81] Nicolas Oury. *Extensionality in the calculus of constructions*, pages 278–293. Springer Berlin Heidelberg, Berlin, Heidelberg, 2005.

[82] Christine Paulin-Mohring. Inductive definitions in the system Coq — rules and properties. In Marc Bezem and Jan Friso Groote, editors, *Typed Lambda*

Calculi and Applications, International Conference on Typed Lambda Calculi and Applications, TLCA '93, Utrecht, The Netherlands, March 16-18, 1993, Proceedings, volume 664 of *Lecture Notes in Computer Science*, pages 328–345. Springer, 1993.

[83] Frank Pfenning and Carsten Schürmann. System description: Twelf — a meta-logical framework for deductive systems. In H. Ganzinger, editor, *Proceedings of the 16th International Conference on Automated Deduction (CADE-16)*, pages 202–206, Trento, Italy, July 1999. Springer-Verlag LNAI 1632.

[84] Brigitte Pientka and Joshua Dunfield. Beluga: A framework for programming and reasoning with deductive systems (system description). In *Proceedings of the 5th International Conference on Automated Reasoning*, IJCAR'10, pages 15–21, Berlin, Heidelberg, 2010. Springer-Verlag.

[85] Gordon Plotkin and Martín Abadi. *A logic for parametric polymorphism*, pages 361–375. Springer Berlin Heidelberg, Berlin, Heidelberg, 1993.

[86] Gordon D. Plotkin. Lambda-definability and logical relations. Memorandum SAI–RM–4, University of Edinburgh, Edinburgh, Scotland, October 1973.

[87] Gordon D. Plotkin. Lambda-definability in the full type hierarchy. In J. P. Seldin and J. R. Hindley, editors, *To H.B. ˜Curry: Essays on Combinatory Logic, Lambda Calculus and Formalism*. Academic Press, London, 1980.

[88] The Univalent Foundations Program. Homotopy type theory: Univalent foundations of mathematics. Technical report, Institute for Advanced Study, 2013.

[89] John C. Reynolds. Types, abstraction and parametric polymorphism. In R. E. A. Mason, editor, *Information Processing 83, Proceedings of the IFIP 9th World Computer Congress, Paris, September 19-23, 1983*, pages 513–523. Elsevier Science Publishers B. V. (North-Holland), Amsterdam, 1983.

[90] Steven Schäfer, Gert Smolka, and Tobias Tebbi. Completeness and decidability of de Bruijn substitution algebra in Coq. In *Proceedings of the 2015 Conference on Certified Programs and Proofs, CPP 2015, Mumbai, India, January 15-17, 2015*, pages 67–73. ACM, 2015.

[91] Michael Shulman. Univalence for inverse diagrams and homotopy canonicity. *Mathematical Structures in Computer Science*, 25:1203–1277, 6 2015. arXiv:1203.3253.

[92] Kristina Sojakova. Higher inductive types as homotopy-initial algebras. *SIGPLAN Not.*, 50(1):31–42, January 2015.

[93] Richard Statman. Completeness, invariance and lambda-definability. *J. Symb. Log.*, 47(1):17–26, 1982.

[94] Thomas Streicher. *Semantics of Type Theory: Correctness, Completeness, and Independence Results.* Birkhauser Boston Inc., Cambridge, MA, USA, 1991.

[95] Thomas Streicher. Investigations into intensional type theory. habilitation thesis, 1993.

[96] William W. Tait. Intensional interpretations of functionals of finite type i. *J. Symbolic Logic*, 32(2):198–212, 06 1967.

[97] The Agda development team. Agda wiki, 2016.

[98] A.S. Troelstra and D. van Dalen. *Constructivism in Mathematics: An Introduction.* Studies in logic and the foundations of mathematics. North-Holland, 1988.

[99] Philip Wadler. Theorems for free! In *Functional Programming Languages and Computer Architecture*, pages 347–359. ACM Press, 1989.