Type Theory in Type Theory (draft)

Thorsten Altenkirch (University of Nottingham)
Ambrus Kaposi (Eötvös Loránd University)

December 19, 2016

Abstract

We present an internal formalisation of a type theory with dependent types in Type Theory using a special case of higher inductive types from Homotopy Type Theory which we call quotient inductive types (QITs). Our formalisation of type theory avoids referring to preterms or a typability relation (but is equivalent to one such definition) and defines directly well typed objects. The elimination principle of the QIT ensures that every construction on the syntax respects the conversion relation. We use the elimination principle to define some models of this theory: the standard model, the setoid model, the presheaf model and the logical predicate interpretation. The work has been formalized in Agda extended with the QIT of the syntax using postulates.

1 Introduction

We would like to reflect the syntax and typing rules of Type Theory in itself. This offers exciting opportunities for typed metaprogramming to support interactive theorem proving. We can also implement extensions of Type Theory by providing a sound interpretation giving rise to a form of template Type Theory. The main idea of our approach is to represent the conversion relation using equality. This paper is an expanded version of our conference paper [7]. In particular we describe more models (the presheaf and setoid models), more type fomers and we add a proof that our syntax is equivalent to a more traditional definition.

Within Type Theory it is straightforward to represent the simply typed λ-calculus as an inductive type where contexts and types are defined as inductive types and terms are given as an inductively defined family indexed by contexts and types (see figure 1 for a definition in idealized Agda).

Here we inductively define Types (Ty) and contexts (Con) which in a de Bruijn setting are just sequences of types. We define the families of variables and terms where variables are *typed de Bruijn indices* and terms are inductively generated from variables using application (app) and abstraction (lam).[1]

In this approach we never define preterms and a typing relation but directly present the typed syntax. This has technical advantages: in typed metaprogramming we only want to deal with typed syntax and it avoids the need to prove subject reduction theorems separately. But more importantly this approach reflects our type-theoretic philosophy that typed objects are first and preterms are designed at a second step with the intention that they should contain enough information so that we can reconstruct the typed objects. Typechecking can be expressed in this view as constructing a partial inverse to the forgetful function which maps typed into untyped syntax (a sort of printing operation).

Naturally, we would like to do the same for the language we are working in, that is we would like to perform typed metaprogramming for a dependently typed language. There are at least two complications:

(1) types, terms and contexts have to be defined mutually but also depend on each other,

(2) due to the conversion rule which allows us to coerce terms along type-equality we have to define the equality mutually with the syntactic typing rules:

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

[1] We are oversimplifying things a bit here: we would really like to restrict operations to those which preserve $\beta\eta$-equality, i.e. work with a quotient of Tm.

$$
\begin{array}{ll}
\mathsf{data}\ \mathsf{Ty} & : \mathsf{Set} \\
\quad \iota & : \mathsf{Ty} \\
\quad - \Rightarrow - & : \mathsf{Ty} \to \mathsf{Ty} \to \mathsf{Ty}
\end{array}
$$

$$
\begin{array}{ll}
\mathsf{data}\ \mathsf{Con} & : \mathsf{Set} \\
\quad \cdot & : \mathsf{Con} \\
\quad -, - & : \mathsf{Con} \to \mathsf{Ty} \to \mathsf{Con}
\end{array}
$$

$$
\begin{array}{ll}
\mathsf{data}\ \mathsf{Var} & : \mathsf{Con} \to \mathsf{Ty} \to \mathsf{Set} \\
\quad \mathsf{zero} & : \mathsf{Var}\,(\Gamma, A)\,A \\
\quad \mathsf{suc} & : \mathsf{Var}\,\Gamma\,A \to \mathsf{Var}\,(\Gamma, B)\,A
\end{array}
$$

$$
\begin{array}{ll}
\mathsf{data}\ \mathsf{Tm} & : \mathsf{Con} \to \mathsf{Ty} \to \mathsf{Set} \\
\quad \mathsf{var} & : \mathsf{Var}\,\Gamma\,A \to \mathsf{Tm}\,\Gamma\,A \\
\quad \mathsf{app} & : \mathsf{Tm}\,\Gamma\,(A \Rightarrow B) \to \mathsf{Tm}\,\Gamma\,A \to \mathsf{Tm}\,\Gamma\,B \\
\quad \mathsf{lam} & : \mathsf{Tm}\,(\Gamma, A)\,B \to \mathsf{Tm}\,\Gamma\,(A \Rightarrow B)
\end{array}
$$

Figure 1: Simply typed λ-calculus

(1) can be addressed by using inductive-inductive definitions [9] (see section 2.1) – indeed doing Type Theory in Type Theory was one of the main motivations behind introducing inductive-inductive types. It seemed that this would also suffice to address (2), since we can define the conversion relation mutually with the rest of the syntax. However, it turns out that the overhead in form of type-theoretic boilerplate is considerable. For example terms depend on both contexts and types, we have to include special constructors which formalize the idea that this is a setoid indexed over two given setoids. Moreover each constructor comes with a congruence rule. In the end the resulting definition becomes unfeasible, almost impossible to write down explicitly but even harder to work with in practice.

This is where Higher Inductive Types [43] come in, because they allow us to define constructors for equalities at the same time as we define elements. Hence we can present the conversion relation just by stating the equality rules as equality constructors. Since we are defining equalities we don't have to formalize any indexed setoid structure or explicitly state congruence rules. This second step turns out to be essential to have a workable internal definition of Type Theory.

Indeed, we only use a rather simple special case of higher inductive types, namely we are only interested in first order equalities and we are ignoring higher equalities. This is what we call Quotient Inductive Types (QITs). From the perspective of Homotopy Type Theory QITs are HITs which are truncated to be sets. The main aspect of HITs we are using is that they allow us to introduce new equalities and constructors at the same time. This has already been exploited in a different way in [43] for the definition of the reals and the constructible hierarchy without having to use the axiom of choice.

1.1 Overview of the Paper

We start by explaining in some detail the type theory we are using as a metatheory and how we implement it using Agda in section 2. In particular we are reviewing inductive-inductive types (section 2.1) and we motivate the importance of QITs as simple HITs (section 2.2). Then, in section 3 we turn to our main goal and define the syntax of a basic type theory with only Π-types and an uninterpreted base type with explicit substitutions. We explain how a general recursor and eliminator can be derived. As a warm-up we use the recursor to define the *standard interpretation* which interprets every piece of syntax by the corresponding metatheoretic construction in section 4. Then we show that our definition of type theory is equivalent to one with preterms and a typing relation in section 5. As real-world examples, we define three interpretations of the syntax: we use the recursor to define the setoid model (section 6) and the presheaf model (section 7) and we use the eliminator for the logical predicate translation of Bernardy [11] (section 8). In section 9 we show how to extend our syntax with Σ types, natural numbers, a universe

and the identity type. We conclude in section 10. The constructions presented in this paper have been formalized in Agda, this is available as supplementary material [6].

1.2 Related Work

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) [24]. Examples of implementations of LF are Twelf [37] and Beluga [38]. Palsberg and coauthors define self-representations of System U [14] and System F_ω [15] using HOAS. In the latter work they also define a self-interpreter.

The work of [22] provides a very good motivation for this paper 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 [33] 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 [18] is the closest predecessor to our work. He represents the typed syntax of simple type theory and proves normalisation using a big-step semantics [3]. He extends this method to dependent type theory [19] 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 very large inductive inductive definition. While this is a remarkable achievement, it is very 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 [21] 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 in [27] where the authors present a mechanism to safely extend 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) [23, 26]. 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 [17] which has been recently used by Coquand to give very concise presentations of Type Theory [12]. 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 [43]. Our metatheory is not homotopy type theory, we only use a special case of higher inductive types called quotient inductive types. These are more general than quotient types [25, 34].

We proved normalisation for the theory described in this paper, see [5].

2 Metatheory

Our metatheory is intensional Martin-Löf type theory with a hierarchy of universes, Π types, a strict equality type, inductive inductive types, and quotient inductive types (QITs) added using axioms and rewrite rules [20]. We will describe these features and the notation through examples in this section.

We use a notation which is very close to that of the functional programming language and a proof assistant Agda [1, 36]. We denote the universes by $\mathsf{Set} = \mathsf{Set}_0, \mathsf{Set}_1, ...$ and write function types as $(x : A) \to B$ or $\forall x.B$. For defining functions, we use λ notation ($\mathsf{d} := \lambda x.t : (x : A) \to B$), Coq-style

notation $(\mathsf{d}\,(x : A) : B := t)$ and Agda's pattern matching notation (declaration $\mathsf{d} : (x : A) \to B$ and implementation $\mathsf{d}\,x := t$).

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 simply $(B \to C) \to (A \to B) \to (A \to C)$ and 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.

We use underscores for arguments of a function that we are not interested in, e.g. the constant function is defined as $\mathsf{const}\,x\,\_ := x$. We write dependent sum types (Σ-types) as $(x : A) \times B$ and projections as $(-)_1$ and $(-)_2$. The keyword record is for defining dependent record types (iterated Σ-types) by listing their fields. Records are automatically modules (separate name spaces), this is why they can be opened using the open keyword. In this case the field names become projection functions. Records and modules can be parameterised (by a telescope of types), this can be used to structure our code. We overload names, e.g. context extension and substitution extension are denoted by the same constructor $-, -$.

The identity type (propositional equality) is denoted $- \equiv -$ and its constructor is refl. We use a strict identity type, that is uniqueness of identity proofs (equivalent to Streicher's axiom K [42]) holds. Transport of a term $u : P\,a$ along an equality $p : a \equiv a'$ is denoted ${}_{p*}u : P\,a'$. We denote $({}_{p*}u) \equiv u'$ by $u \equiv^{p} u'$. We write ap for congruence, that is $\mathsf{ap}\,f\,p : f\,a \equiv f\,a'$ if $p : a \equiv a'$. We write $- \centerdot -$ for transitivity and $-^{-1}$ for symmetry of equality.

We assume functional extensionality which follows in any case from the presence of QITs. The theory poses a canonicity problem, i.e. can all closed term of type $\mathbb{N}$ be reduced to numerals, which can be adressed using techniques developed in the context of observational type theory [8]. Recent work [12] suggest that also the harder problem of canonicity in the presence of univalence can be addressed.

2.1 Inductive inductive types

One central construction in Type Theory are *inductive types*, where types are specified using constructors - see the definition of types and contexts in figure 1. In practical dependently typed programming we use pattern matching - however it is good to know that this can be replaced by adding just one elimination constant to each inductive type we are defining [32]. Here we differentiate between the *recursor* which enables us to define non-dependent functions and the eliminator which allows the definition of dependent functions. The latter corresponds logically to an induction principle for the given type. Note that the recursor is just a special case of the eliminator. As an example consider the recursor and the eliminator for the inductive type Ty defined in figure 1:

$$
\begin{aligned}
\mathsf{RecTy}\ :\ & (\mathsf{Ty}^{\mathsf{M}} : \mathsf{Set}) \\
& (\iota^{\mathsf{M}} : \mathsf{Ty}^{\mathsf{M}}) \\
& (\Rightarrow^{\mathsf{M}} : \mathsf{Ty}^{\mathsf{M}} \to \mathsf{Ty}^{\mathsf{M}} \to \mathsf{Ty}^{\mathsf{M}}) \\
& \to \mathsf{Ty} \to \mathsf{Ty}^{\mathsf{M}} \\
\mathsf{RecTy}\,\mathsf{Ty}^{\mathsf{M}}\,\iota^{\mathsf{M}}\,\Rightarrow^{\mathsf{M}}\,\iota &= \iota^{\mathsf{M}} \\
\mathsf{RecTy}\,\mathsf{Ty}^{\mathsf{M}}\,\iota^{\mathsf{M}}\,\Rightarrow^{\mathsf{M}}\,(A \Rightarrow B) &= \Rightarrow^{\mathsf{M}}\,(\mathsf{RecTy}\,\mathsf{Ty}^{\mathsf{M}}\,\iota^{\mathsf{M}}\,\Rightarrow^{\mathsf{M}}\,A)\,(\mathsf{RecTy}\,\mathsf{Ty}^{\mathsf{M}}\,\iota^{\mathsf{M}}\,\Rightarrow^{\mathsf{M}}\,B) \\
\mathsf{ElimTy}\ :\ & (\mathsf{Ty}^{\mathsf{M}} : \mathsf{Ty} \to \mathsf{Set}) \\
& (\iota^{\mathsf{M}} : \mathsf{Ty}^{\mathsf{M}}\,\iota) \\
& \big(\Rightarrow^{\mathsf{M}} : \{A : \mathsf{Ty}\} \to \mathsf{Ty}^{\mathsf{M}}\,A \to \{B : \mathsf{Ty}\} \to \mathsf{Ty}^{\mathsf{M}}\,B \to \mathsf{Ty}^{\mathsf{M}}\,(A \Rightarrow B)\big) \\
& \to (C : \mathsf{Ty}) \to \mathsf{Ty}^{\mathsf{M}}\,C \\
\mathsf{ElimTy}\,\mathsf{Ty}^{\mathsf{M}}\,\iota^{\mathsf{M}}\,\Rightarrow^{\mathsf{M}}\,\iota &= \iota^{\mathsf{M}} \\
\mathsf{ElimTy}\,\mathsf{Ty}^{\mathsf{M}}\,\iota^{\mathsf{M}}\,\Rightarrow^{\mathsf{M}}\,(A \Rightarrow B) &= \Rightarrow^{\mathsf{M}}\,(\mathsf{ElimTy}\,\mathsf{Ty}^{\mathsf{M}}\,\iota^{\mathsf{M}}\,\Rightarrow^{\mathsf{M}}\,A)\,(\mathsf{ElimTy}\,\mathsf{Ty}^{\mathsf{M}}\,\iota^{\mathsf{M}}\,\Rightarrow^{\mathsf{M}}\,B)
\end{aligned}
$$

The type/dependent type we use in the recursor/eliminator (Ty^{M}) we call the *motive* and the functions corresponding to the constructors (ι^{M}, $\Rightarrow^{\mathsf{M}}$) are the *methods*. The motive and the methods of the recursor are the *algebras* of the corresponding signature functor. The motive Ty^{M} is a *family indexed* over Ty and the methods are *fibers* of the family over the constructors.

We can also define dependent families of types inductively – examples are Var and Tm in figure 1.

We may extend this to mutual inductive types or mutual inductive dependent types. We define mutual inductive types by first declaring the types and then separately listing the constructors for each. An example is the definition of the predicates Odd and Even on natural numbers.

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

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.

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

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

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 [35] 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 Even-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.

The eliminator for an IIT needs as many motives as the number of constituent types and a method for each constructor. In our example case 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.

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

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).

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

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 8.6.

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

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.

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

The categorical semantics of inductive inductive types has been explored in [9]. From a computational point of view IITs are unproblematic and they are supported by Agda.

2.2 Quotient inductive types

Higher inductive types come from homotopy type theory (chapter 6 of [43]). 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 segment which adds an element to the identity type for 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 segment. The method corresponding to segment ensures that the things which left and right are mapped to are also equal. Note that in the case of the dependent eliminator this equality lives over the constructor segment. We list the recursor and the eliminator below.

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

The computation rules for 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 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. 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 meen 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 paper. It also follows from the existence of the interval quotient inductive type I. For details, see the formalisation [6].

QITs are different from quotient types [25, 29]: for quotient types, one needs to define the type and the equivalence relation in separate steps. However this is sometimes not possible as in our main use case of defining the syntax of type theory (see chapter 3). There are two more such examples in [43]: the constructible 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]

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

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.

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

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 $\mathsf{perm}^{\mathsf{R}}$ 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.

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

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.

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

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

[2] A quotient type can be seen as a higher inductive type with the given constructors.

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 there is no way of expressing such infinite definitions 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.

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

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

Quotient inductive types are a convenient way to define generalised algebraic theories [17]. 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 [16, 31, 41]. Other special cases of HITs have been proposed and used e.g. in chapter 6 of [43] but a general theory of HITs is still missing.

QIITs can be added to Agda by postulating all the constructors and adding the computation rules as rewrite rules [20]. This avoids the problems with previous approaches such as [30] using inductive types and only postulating the equality constructors.

3 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.

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

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 substitution calculus, and then we add rules for different type formers separately.

3.1 The substitution calculus

The constructors of the substitution calculus are given below.

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

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.

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 an 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.

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

  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. The other direction ($\pi_1\circ$ and $\pi_2[]$) can be proven.

Using our categorically inspired syntax we can derive more traditional syntactic constructions such as variables expressed as typed De Bruijn indices. 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 Peano 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 A base type and family

We add an uninterpreted base type U and a family El over this type. El is the only way in our basic syntax where terms leak into types. U and El can be viewed as the dependently typed correspondent to the usual base type ι of simple type theory with no constructors: they are the simplest type formers which make the theory interesting.

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.

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

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 $\mathsf{U}[]$.

3.3 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 $\Delta.A$ 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.

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

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

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

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

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

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.

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

3.4 The elimination principle

When we define a function from the above syntax, we need to use the eliminator. This is defined analogously to the examples in secion 2. The eliminator has 4 motives corresponding to what Con, Ty, Tms and Tm get mapped to and one method for each constructor including the equality constructors. The methods for point constructors are the elements of the motives to which the constructor is mapped. The methods for the equality constructors demonstrate soundness, that is, the semantic constructions respect the syntactic equalities.

The motives and methods for the recursor (non dependent eliminator) collected together form a model of type theory, they are equivalent to Dybjer's Categories with Families [23]. We list some of the motives and methods for the dependent eliminator below. For a complete presentation and an algorithm for

deriving these from the constructors, see [28].

$$
\begin{aligned}
&\mathsf{Con}^{\mathsf{M}} && : \mathsf{Con} \to \mathsf{Set}\\
&\mathsf{Ty}^{\mathsf{M}} && : (\mathsf{Con}^{\mathsf{M}}\,\Gamma) \to \mathsf{Ty}\,\Gamma \to \mathsf{Set}\\
&\mathsf{Tms}^{\mathsf{M}} && : (\mathsf{Con}^{\mathsf{M}}\,\Gamma) \to (\mathsf{Con}^{\mathsf{M}}\,\Delta) \to \mathsf{Tms}\,\Gamma\,\Delta \to \mathsf{Set}\\
&\mathsf{Tm}^{\mathsf{M}} && : (\Gamma^{\mathsf{M}} : \mathsf{Con}^{\mathsf{M}}\,\Gamma) \to \mathsf{Ty}^{\mathsf{M}}\,\Gamma^{\mathsf{M}}\,A \to \mathsf{Tm}\,\Gamma\,A \to \mathsf{Set}\\
&\cdot^{\mathsf{M}} && : \mathsf{Con}^{\mathsf{M}}\,\cdot\\
&-,^{\mathsf{M}}- && : (\Gamma^{\mathsf{M}} : \mathsf{Con}^{\mathsf{M}}\,\Gamma) \to \mathsf{Ty}^{\mathsf{M}}\,\Gamma^{\mathsf{M}}\,A \to \mathsf{Con}^{\mathsf{M}}\,(\Gamma, A)\\
&\mathsf{id}^{\mathsf{M}} && : \mathsf{Tms}^{\mathsf{M}}\,\Gamma^{\mathsf{M}}\,\Gamma^{\mathsf{M}}\,\mathsf{id}\\
&-\circ^{\mathsf{M}}- && : \mathsf{Tms}^{\mathsf{M}}\,\Theta^{\mathsf{M}}\,\Delta^{\mathsf{M}}\,\sigma \to \mathsf{Tms}^{\mathsf{M}}\,\Gamma^{\mathsf{M}}\,\Theta^{\mathsf{M}}\,\nu \to \mathsf{Tms}^{\mathsf{M}}\,\Gamma^{\mathsf{M}}\,\Delta^{\mathsf{M}}\,(\sigma \circ \nu)\\
&\circ\mathsf{id}^{\mathsf{M}} && : \sigma^{\mathsf{M}} \circ^{\mathsf{M}} \mathsf{id}^{\mathsf{M}} \equiv^{\circ\mathsf{id}} \sigma^{\mathsf{M}}\\
&\pi_2\beta^{\mathsf{M}} && : \pi_2^{\mathsf{M}}\,(\rho^{\mathsf{M}},^{\mathsf{M}} t^{\mathsf{M}}) \equiv^{\pi_1\beta^{\mathsf{M}},\pi_2\beta} t^{\mathsf{M}}
\end{aligned}
$$

Note that the method equality $\circ\mathsf{id}^{\mathsf{M}}$ lives over the constructor $\circ\mathsf{id}$ while the method equality $\pi_2\beta^{\mathsf{M}}$ lives both over the method equality $\pi_1\beta^{\mathsf{M}}$ and the equality constructor $\pi_2\beta$.

There are four eliminators for the four constituent types. These are understood in the presence of all the motives and methods given above.

$$
\begin{aligned}
&\mathsf{ElimCon} && : (\Gamma : \mathsf{Con}) && \to \mathsf{Con}^{\mathsf{M}}\,\Gamma\\
&\mathsf{ElimTy} && : (A : \mathsf{Ty}\,\Gamma) && \to \mathsf{Ty}^{\mathsf{M}}\,(\mathsf{ElimCon}\,\Gamma)\,A\\
&\mathsf{ElimTms} && : (\sigma : \mathsf{Tms}\,\Gamma\,\Delta) && \to \mathsf{Tms}^{\mathsf{M}}\,(\mathsf{ElimCon}\,\Gamma)\,(\mathsf{ElimCon}\,\Delta)\,\sigma\\
&\mathsf{ElimTm} && : (t : \mathsf{Tm}\,\Gamma\,A) && \to \mathsf{Tm}^{\mathsf{M}}\,(\mathsf{ElimCon}\,\Gamma)\,(\mathsf{ElimTy}\,A)\,t
\end{aligned}
$$

We have the usual β computation rules such as the following.

$$
\begin{aligned}
&\mathsf{ElimCon}\,(\Gamma, A) = \mathsf{ElimCon}\,\Gamma,^{\mathsf{M}} \mathsf{ElimTy}\,A\\
&\mathsf{ElimTms}\,(\sigma \circ \nu) = \mathsf{ElimTms}\,\sigma \circ^{\mathsf{M}} \mathsf{ElimTms}\,\nu
\end{aligned}
$$

4 The standard interpretation

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. That is, our interpretation is declared by the following motives. We use pattern matching notation for readability e.g. we write $\mathsf{Ty}^{\mathsf{M}}\,[\![\Gamma]\!] := t$ instead of $\mathsf{Ty}^{\mathsf{M}} := \lambda[\![\Gamma]\!].t$. Note also that we use the double square brackets as part of the variable names, there is no function $[\![-]\!]$.

$$
\begin{aligned}
&\mathsf{Con}^{\mathsf{M}} && := \mathsf{Set}\\
&\mathsf{Ty}^{\mathsf{M}}\,[\![\Gamma]\!] && := [\![\Gamma]\!] \to \mathsf{Set}\\
&\mathsf{Tms}^{\mathsf{M}}\,[\![\Gamma]\!]\,[\![\Delta]\!] && := [\![\Gamma]\!] \to [\![\Delta]\!]\\
&\mathsf{Tm}^{\mathsf{M}}\,[\![\Gamma]\!]\,[\![A]\!] && := (\alpha : [\![\Gamma]\!]) \to [\![A]\!]\,\alpha
\end{aligned}
$$

The methods for the substitution calculus are the following. Context extension is interpreted by dependent sum, substitution composition by function composition, projections by projections. The interpretation

of the equalities are all refl because the two sides are actually convertible in the metatheory.

$$
\begin{aligned}
&\cdot^{\mathsf{M}} && := \top \\
&[\![\Gamma]\!],^{\mathsf{M}}\, [\![A]\!] && := (\alpha : [\![\Gamma]\!]) \times [\![A]\!]\, \alpha \\
&[\![A]\!][[\![\sigma]\!]]^{\mathsf{M}}\, \alpha && := [\![A]\!]\, ([\![\sigma]\!]\, \alpha) \\
&[\![\sigma]\!] \circ^{\mathsf{M}} [\![\nu]\!]\alpha && := [\![\sigma]\!]\, ([\![\nu]\!]\, \alpha) \\
&\mathsf{id}^{\mathsf{M}} && \alpha := \alpha \\
&\epsilon^{\mathsf{M}} && \_ := \mathsf{tt} \\
&[\![\sigma]\!],^{\mathsf{M}}\, [\![t]\!] && \alpha := ([\![\sigma]\!]\, \alpha, [\![t]\!]\, \alpha) \\
&{\pi_1}^{\mathsf{M}}\, [\![\sigma]\!] && \alpha := ([\![\sigma]\!]\, \alpha)_1 \\
&[\![t]\!][[\![\sigma]\!]]^{\mathsf{M}} && \alpha := [\![t]\!]\, ([\![\sigma]\!]\, \alpha) \\
&{\pi_2}^{\mathsf{M}}\, [\![\sigma]\!] && \alpha := ([\![\sigma]\!]\, \alpha)_2 \\
&[][]^{\mathsf{M}} && \alpha := \mathsf{refl} \\
&... &&
\end{aligned}
$$

To interpret the base type and family, we parameterise the model by $[\![\mathsf{U}]\!] : \mathsf{Set}$ and $[\![\mathsf{El}]\!] : [\![\mathsf{U}]\!] \to \mathsf{Set}$. These provide the interpretations of U and El, respectively.

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

In the methods for function space, the interpretation of Π is metatheoretic dependent function space, the intepretation of lam is metatheoretic lambda etc.

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

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).

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

It should 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).

5 Equivalence to an extrinsic definition

6 Setoid model

In this section we define the setoid model [2] for the theory given in section 3.

A setoid is a set together with an equivalence relation. The relation is propositional, that is, all witnesses of relatedness are equal. A setoid is given by an element of the record Setoid.

$$
\begin{aligned}
&\mathsf{record}\ \mathsf{Setoid}\\
&\quad |-| \qquad : \mathsf{Set}\\
&\quad - \sim_{-} - : |-| \to |-| \to \mathsf{Set}\\
&\quad \mathsf{prop}_{-} \quad : (p\,q : \gamma \sim \gamma') \to p \equiv q\\
&\quad \mathsf{refl}_{-} \quad : \gamma \sim \gamma\\
&\quad {-^{-1}}_{-} \quad : \gamma \sim \gamma' \to \gamma' \sim \gamma\\
&\quad - \centerdot_{-} - \ : \gamma \sim \gamma' \to \gamma' \sim \gamma'' \to \gamma \sim \gamma''
\end{aligned}
$$

Given a setoid Γ, $|\Gamma|$ is the carrier set, $\sim_\Gamma$ is a relation which is propositional by prop_Γ and refl_Γ, ${-^{-1}}_\Gamma$ and $-\centerdot_\Gamma-$ express reflexivity, symmetry and transitivity of the relation. If Γ is clear from the context, we omit it when writing the projection, e.g. we might write prop instead of prop_Γ.

Families of setoids over a setoid are given by the following definition.

$$
\begin{aligned}
&\mathsf{record}\ \mathsf{FamSetoid}\ (\Gamma : \mathsf{Setoid})\\
&\quad |-| \qquad : |\Gamma| \to \mathsf{Set}\\
&\quad - \sim^{-}_{-} - : \gamma \sim_\Gamma \gamma' \to |-|\,\gamma \to |-|\,\gamma' \to \mathsf{Set}\\
&\quad \mathsf{prop}_{-} \quad : (r : a \sim^{p} a')(s : a \sim^{q} a') \to r \equiv^{\mathsf{prop}_\Gamma\ p\ q} s\\
&\quad \mathsf{refl}_{-} \quad : a \sim^{\mathsf{refl}_\Gamma\ \gamma} a\\
&\quad {-^{-1}}_{-} \quad : a \sim^{p} a' \to a' \sim^{p^{-1_\Gamma}} a\\
&\quad - \centerdot_{-} - \ : a \sim^{p} a' \to a' \sim^{q} a'' \to a \sim^{p \centerdot_\Gamma q} a''
\end{aligned}
$$

Note that all the fields depend on the corresponding fields of the setoid over which the family is defined.

A setoid morphism is given by the following definition.

$$
\begin{aligned}
&\mathsf{record}\ (\Gamma : \mathsf{Setoid}) \dot{\to} (\Delta : \mathsf{Setoid})\\
&\quad |-| \quad : |\Gamma| \to |\Delta|\\
&\quad \mathsf{resp}_{-} : \gamma \sim_\Gamma \gamma' \to |-|\,\gamma \sim_\Delta |-|\,\gamma'
\end{aligned}
$$

Given a setoid morphism $\sigma : \Gamma \dot{\to} \Delta$, we have a function $|\sigma| : |\Gamma| \to |\Delta|$ and a proof resp_σ which says that $|\sigma|$ respects the equivalence relations.

A section of setoids is defined as follows

$$
\begin{aligned}
&\mathsf{record}\ (\Gamma : \mathsf{Setoid}) \overset{\mathsf{s}}{\to} (A : \mathsf{FamSetoid}\,\Gamma)\\
&\quad |-| \quad : (\gamma : |\Gamma|) \to |A|\,\gamma\\
&\quad \mathsf{resp}_{-} : (p : \gamma \sim_\Gamma \gamma') \to |-|\,\gamma \sim^{p}_{A} |-|\,\gamma'
\end{aligned}
$$

Given a section t, we have a function $|t| : (\gamma : |\Gamma|) \to |A|\,\gamma$ and a proof that it respects the equivalence relations. Note that $\sim_A$ depends on an element of $\sim_\Gamma$.

Now we can give the declaration of the model.

$$
\begin{aligned}
&\mathsf{Con} \ := \mathsf{Setoid}\\
&\mathsf{Ty} \ := \mathsf{FamSetoid}\\
&\mathsf{Tms} := \dot{\to}\\
&\mathsf{Tm} \ := \overset{\mathsf{s}}{\to}
\end{aligned}
$$

Now we describe the interpretations of the substitution calculus.

The empty context is the trivial setoid.

$$
\begin{aligned}
&|\cdot| \qquad := \top\\
&\gamma \sim_{\cdot} \gamma' := \top\\
&\mathsf{prop}_{\cdot\ \_} \ := \mathsf{tt}\\
&\mathsf{refl}_{\cdot\ \_} \ := \mathsf{tt}\\
&{\_^{-1}}_{\cdot} \qquad := \mathsf{tt}\\
&\_ \centerdot_{\cdot\ \_} \ := \mathsf{tt}
\end{aligned}
$$

Context extension is the dependent sum of the two setoids, all the operations are pointwise. We only list the first two fields.

$$
\begin{aligned}
&|\Gamma, A| && := (\gamma : |\Gamma|) \times |A|\,\gamma \\
&(\gamma, a) \sim_{\Gamma,A} (\gamma', a') && := (p : \gamma \sim_\Gamma \gamma') \times a \sim^p_A a'
\end{aligned}
$$

Substitution of types is given by applying the setoid morphism on the interpretation of the context. To define the relation corresponding to a substituted setoid, we need that the setoid morphism respects the relation.

$$
\begin{aligned}
&|A[\sigma]|\,\gamma && := |A|\,(|\sigma|\,\gamma) \\
&a \sim^p_{A[\sigma]} a' && := a \sim^{\mathsf{resp}_\sigma\, p}_A a'
\end{aligned}
$$

For more details, see the formalisation.

7 Presheaf model

Presheaf models [26] are the proof-relevant versions of Kripke semantics for intuitionistic logic. They are used to model guarded type theory [13], parametricity [10], the local state monad [39] or univalence **??**.

7.1 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.

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

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 $\sigma : \Gamma \dot{\to} \Delta$. 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 is given by the following data: given $\alpha : \Gamma\, I$, a set $A_I\,\alpha$ and given $f : \mathcal{C}(J, I)$, a function $A\, f : A_I\,\alpha \to A_J\,(\Gamma\, f\,\alpha)$. 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 $\sigma : A \overset{\mathsf{N}}{\to} B$. 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 from a presheaf Γ to a family of presheaves A over Γ is denoted $t : \Gamma \overset{\mathsf{s}}{\to} A$. 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 $f : \mathcal{C}(J, I)$.

7.2 Motives

The presheaf model is parameterised over a category $\mathcal{C}$. We use the recursor to define the model, the motives are the following.

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

Note that here $[\![-]\!]$ is not a function, it is just part of the variable names. The interpretation of $\Gamma : \mathsf{Con}$ is a contravariant presheaf $[\![\Gamma]\!]$ over $\mathcal{C}$. The interpretation of a type $A : \mathsf{Ty}\,\Gamma$ is a family of presheaves $[\![A]\!]$ over $[\![\Gamma]\!]$. The interpretation of a subsitution $\rho : \mathsf{Tms}\,\Delta\,\Gamma$ is a natural tranformation $[\![\rho]\!]$ between the presheaves $[\![\Delta]\!]$ and $[\![\Gamma]\!]$. The interpretation of a term $t : \mathsf{Tm}\,\Gamma\,A$ is a section $[\![t]\!] : [\![\Gamma]\!] \overset{\mathsf{s}}{\to} [\![A]\!]$.

7.3 Methods for the substitution calculus

The methods for contexts create presheaves, we only spell out the action on objects. The empty context is the constant unit presheaf, context extension is pointwise.

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

Substituted types are interpreted using the interpretation of the substitution on the environment.

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

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

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

The interpretations of term formers are listed below omitting the naturality proofs.

$$\begin{aligned}
[\![t]\!][[\![\rho]\!]]^{\mathsf{M}}\,\alpha &:= [\![t]\!]\,([\![\rho]\!]\,\alpha)\\
{\pi_2}^{\mathsf{M}}\,[\![t]\!]\,\alpha &:= ([\![\rho]\!]\,\alpha)_2
\end{aligned}$$

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

7.4 Methods for the base type and family

We parameterise the presheaf model by a presheaf $[\![\mathsf{U}]\!]$ and a family of presheaves $[\![\mathsf{El}]\!]$. These interpret U and El. The action on objects is the following. Note that U^{M} does not depend on the environment.

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

We omit the proofs of $\mathsf{U}[]^{\mathsf{M}}$ and $\mathsf{El}[]^{\mathsf{M}}$, it is easy to verify them.

7.5 Methods for the function space

The interpretation of Π is the dependent presheaf exponential which consists of a function map together with a compatibility condition.

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

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 δ, we need to transport this along compP^{-1} to get an element of type $[\![A]\!]\,([\![\Gamma]\!]\,(g \circ f)\,\alpha)$. Now we can apply $\delta\,(g \circ f)$ on it.

The interpretation of lam and app are the following (omitting the naturality proofs and the compatibility condition for $\mathsf{lam}^{\mathsf{M}}$).

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

The equality methods for functions can be verified easily, for details see the formal development.

8 The logical predicate interpretation

In this section we introduce logical predicates informally, then present the formalisation. This is a real-world example of the usefullness of our representation of the syntax; we express logical relations as a syntactic translation following [11] so we define a mapping from the syntax to the syntax itself. This could be useful in connection with metaprogramming: using a quoting mechanism it could provide a way of automatically deriving free theorems [44] for functions defined in a dependently typed programming language.

8.1 Logical relations for dependent types

Logical relations were introduced in computer science by Reynolds [40] for expressing the idea of representation-independence in the context of the polymorphic λ-calculus. Reynold's abstraction theorem (also called parametricity or the fundamental theorem of the logical relation) states that logically related interpretations of a term are logically related at the relation generated by the type of the term. This was later extended to dependent types by [11]: type theory is expressive enough to express the parametricity statement of its own terms — the logic in which the logical relations are expressed can be the theory itself. Contexts are interpreted as syntactic contexts, types as types in the interpretation of their context and terms as witnesses of the interpretation of their types.

A simple example of using parametricity can be described for the following term t.

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

We can view t as a program of type A importing a library which 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 use x. This can be made precise by unary parametricity: for our example it says that if there is a predicate on A and x respects this predicate, then t will also respect it. We set the predicate on A to $A^M\,y := (y \equiv x)$ and observe that this predicate holds for x, as $\mathsf{refl} : A^M\,x$. Now parametricity tells us that the predicate also holds for t, that is, $t \equiv x$.

First we describe the logical predicate interpretation which gives us the above result for an informal type theory with named variables, universes a la Russel, Π types and one universe. The syntax of contexts, terms and types is the following:

$$
\begin{aligned}
\Gamma \qquad &::= \cdot \mid \Gamma, x : A\\
t, u, A, B \;&::= x \mid \mathsf{Set} \mid \Pi(x : A).B \mid \lambda x.t \mid t\,u
\end{aligned}
$$

The typing rules are standard.

We define the unary logical predicate operation $-^{\mathsf{P}}$ on the syntax following [11]. This takes types to their corresponding logical predicates, contexts (lists of types) to lists of related types and terms to witnesses of relatedness in the predicate corresponding to their types. We define $-^{\mathsf{P}}$ by first giving its typing rules for contexts, types and terms (the second rules will become an instance of the third one).

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

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.

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

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 Set is predicate space. This makes the fixpoint of the rule for terms at Set typecheck: $\mathsf{Set}^{\mathsf{P}} : \mathsf{Set}^{\mathsf{P}}\, \mathsf{Set}$ and $\mathsf{Set}^{\mathsf{P}}\, \mathsf{Set} = \mathsf{Set} \to \mathsf{Set}$. The logical predicate holds for a function f 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.

In addition, we need to check whether this definition typechecks (the left and right hand sides need to have convertible types) and respects the conversion rules. We show that this is the case for the formal version in the next subsection.

The example now can be derived using parametricity and substitution.[3]

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

8.2 Motives

Now we turn our attention to the formal development. First we define the motives of the dependent 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.

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

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

[3] We can use Leibniz equality as $\equiv$ or extend the theory with the identity type. In the latter case we need to extend the logical predicate interpretation to the identity type, see [28].

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.

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

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

$$\begin{array}{ccc} |\Gamma^M| & \xrightarrow{\mathsf{Pr}\,\Gamma^M} & \Gamma \\ {\scriptstyle |\sigma^M|}\downarrow & & \downarrow{\scriptstyle \sigma} \\ |\Delta^M| & \xrightarrow{\mathsf{Pr}\,\Delta^M} & \Delta \end{array}$$

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 $t[\mathsf{Pr}\,\Gamma^M]$.

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

8.3 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.

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

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.

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

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. We use equational reasoning notation

for proving the equality.

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

The remaining methods for the point constructors are given by the following definitions, omitting the equality parts and some transports in the case of $-[-]^M$ and π_2^{M}. For these details, see [28].

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

We only state the types for the equality methods for types, they can be all proved by simple equality reasoning.

$$
\begin{aligned}
&[][]^{\mathsf{M}} : A^M[\sigma^M]^{\mathsf{M}}[\nu^M]^{\mathsf{M}} \equiv A^M\big[(|\sigma^M| \circ |\nu^M|) \uparrow A[\mathsf{Pr}]\big] \\
&[\mathsf{id}]^{\mathsf{M}} : A^M[\mathsf{id}^M] \equiv A^M
\end{aligned}
$$

To prove that two elements of the record $\mathsf{Tms}^{\mathsf{M}}$ are equal it is enough to prove that the fields $|-|$ are equal because the other fields will be equal by K. In the following definitions, we only write down the main steps and omit some details.

$$
\begin{aligned}
\circ\circ^{\mathsf{M}} &: (|\sigma^M| \circ |\nu^M|) \circ |\delta^M| \equiv |\sigma^M| \circ (|\nu^M| \circ |\delta^M|) && := \circ\circ \\
\mathsf{id}\circ^{\mathsf{M}} &: \mathsf{id} \circ |\sigma^M| \equiv |\sigma^M| && := \mathsf{id}\circ \\
\circ\mathsf{id}^{\mathsf{M}} &: |\sigma^M| \circ \mathsf{id} \equiv |\sigma^M| && := \circ\mathsf{id} \\
\epsilon\eta^{\mathsf{M}} &: \{\sigma^M : \mathsf{Tms}^{\mathsf{M}}\,\Gamma^M \mathbin{.^{\mathsf{M}}} \sigma\} \to |\sigma^{\mathsf{M}}| \equiv \epsilon && := \epsilon\eta \\
\pi_1\beta^{\mathsf{M}} &: \pi_1\,\big(\pi_1\,(|\sigma^M|, t[\mathsf{Pr}\,\Gamma^M], t^M)\big) \equiv |\sigma^M| && := \mathsf{ap}\,\pi_1\,\pi_1\beta \mathbin{\blacksquare} \pi_1\beta \\
\pi\eta^{\mathsf{M}} &: \big(\pi_1\,(\pi_1\,|\sigma^M|), (\pi_2\,\sigma)[\mathsf{Pr}\,\Gamma^M], \pi_2\,|\sigma^M|\big) \equiv |\sigma^M| && := \mathsf{PrNat}\,\sigma^M \mathbin{\blacksquare} \pi\eta \mathbin{\blacksquare} \pi\eta \\
,\circ^{\mathsf{M}} &: |\nu^M ,^{\mathsf{M}} t^M| \circ^{\mathsf{M}} \sigma^M \equiv (\nu^M \circ^{\mathsf{M}} \sigma^M) ,^{\mathsf{M}} t^M[\sigma^M]^{\mathsf{M}} && := ,\circ \mathbin{\blacksquare} ,\circ
\end{aligned}
$$

The single equality method for terms is given as follows.

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

8.4 Methods for the base type and family

We parameterise the logical predicate interpretation 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.

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

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

$$\mathsf{U}^\mathsf{M} : \mathsf{Ty}\,(|\Gamma^M|, \mathsf{U}) := \bar{\mathsf{U}}[\epsilon, \mathsf{vz}]$$
$$\mathsf{El}^\mathsf{M}\,\{\hat{A} : \mathsf{Tm}\,\Gamma\,\mathsf{U}\}\,\big(\hat{A}^M : \mathsf{Tm}\,|\Gamma^M|\,(\bar{\mathsf{U}}[< \hat{A}[\mathsf{Pr}\,\Gamma^M] >])\big) : \mathsf{Ty}\,(|\Gamma^M|, \mathsf{El}\,\hat{A}[\mathsf{Pr}\,\Gamma^M])$$
$$:= \bar{\mathsf{El}}\big[\epsilon, \hat{A}[\mathsf{Pr}\,\Gamma^M][\mathsf{wk}], \mathsf{vz}, \hat{A}^M[\mathsf{wk}]\big]$$

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 substituion 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). The equalities are proven by laws of the substitution calculus and $\mathsf{PrNat}\,\sigma^M$.

$$\mathsf{U}[]^\mathsf{M} : \ \mathsf{U}^\mathsf{M}[\sigma^M]^\mathsf{M} = \hat{\mathsf{U}}[\epsilon, \mathsf{vz}][|\sigma^M| \uparrow] \equiv \hat{\mathsf{U}}[\epsilon \circ (|\sigma^M| \uparrow), \mathsf{vz}] \equiv \hat{\mathsf{U}}[\epsilon, \mathsf{vz}] = \mathsf{U}^\mathsf{M}$$
$$\mathsf{El}[]^\mathsf{M} : \ (\mathsf{El}^\mathsf{M}\,\hat{A}^M)[\sigma^M]^\mathsf{M} = \bar{\mathsf{El}}\big[\epsilon, \hat{A}[\mathsf{Pr}][\mathsf{wk}], \mathsf{vz}, \hat{A}^M[\mathsf{wk}]\big]\,\big[|\sigma^M| \uparrow\big]$$
$$\equiv \bar{\mathsf{El}}\big[\epsilon, \hat{A}[\mathsf{Pr} \circ |\sigma|^M][\mathsf{wk}], \mathsf{vz}, \hat{A}^M[|\sigma^M|][\mathsf{wk}]\big] \equiv \bar{\mathsf{El}}\big[\epsilon, \hat{A}[\sigma \circ \mathsf{Pr}][\mathsf{wk}], \mathsf{vz}, \hat{A}^M[|\sigma^M|][\mathsf{wk}]\big]$$
$$= \mathsf{El}^\mathsf{M}\,(\hat{A}^M[\sigma^M]^\mathsf{M})$$

8.5 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, 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. 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 $|\sigma^M \uparrow^\mathsf{M}| \equiv |\sigma^M| \uparrow\uparrow$ (note that $- \uparrow^\mathsf{M} -$ was defined

when defining the eliminator: it is the semantic counterpart of $\uparrow$).

$$
\begin{aligned}
\Pi[]^{\mathsf{M}} :\ & (\Pi^{\mathsf{M}}\, A^M\, B^M)[\sigma^M]^{\mathsf{M}} \\
=& \Big(\Pi\,(A[\mathsf{Pr}\circ\mathsf{wk}])\,\big(\Pi\,(A^M[\mathsf{wk}\uparrow])\,(B^M[\mathsf{wk}\uparrow\uparrow, 2\$1])\big)\Big)[|\sigma^M|\uparrow] \\
& && (\Pi[]) \\
\equiv& \Big(\Pi\,(A[\mathsf{Pr}\circ\mathsf{wk}][|\sigma^M|\uparrow])\,\big(\Pi\,(A^M[\mathsf{wk}\uparrow][|\sigma^M|\uparrow\uparrow]) \\
& \qquad (B^M[\mathsf{wk}\uparrow\uparrow, 2\$1][|\sigma^M|\uparrow\uparrow\uparrow])\big)\Big) \\
& && (\text{substitution calculus}) \\
\equiv& \Big(\Pi\,(A[\mathsf{Pr}\circ|\sigma^M|][\mathsf{wk}])\,\big(\Pi\,(A^M[|\sigma^M|\circ\mathsf{wk}^2, 0]) \\
& \qquad (B^M[|\sigma^M|\circ\mathsf{wk}^3, 1, 0, 2\$1])\big)\Big) \\
& && (\mathsf{PrNat}) \\
\equiv& \Big(\Pi\,(A[\sigma\circ\mathsf{Pr}][\mathsf{wk}])\,\big(\Pi\,(A^M[|\sigma^M|\circ\mathsf{wk}^2, 0]) \\
& \qquad (B^M[|\sigma^M|\circ\mathsf{wk}^3, 1, 0, 2\$1])\big)\Big) \\
& && (\text{substitution calculus}) \\
\equiv& \Pi\,(A[\sigma][\mathsf{Pr}\circ\mathsf{wk}])\,\big(\Pi\,(A^M[|\sigma^M|\uparrow][\mathsf{wk}\uparrow]) \\
& \qquad (B^M[|\sigma^M|\circ\mathsf{wk}^3, 2, 1, 0][\mathsf{wk}\uparrow\uparrow, 2\$1])\big) \\
=& \Pi^{\mathsf{M}}(A^M[|\sigma^M|\uparrow])\,(B^M[(|\sigma^M|\circ\mathsf{wk}^2, 1, 0)\uparrow]) \\
=& \Pi^{\mathsf{M}}(A^M[\sigma^M]^{\mathsf{M}})\,(B^M[\sigma^M\uparrow^{\mathsf{M}}]^{\mathsf{M}})
\end{aligned}
$$

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.

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

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

$$
\mathsf{lam}[]^{\mathsf{M}} : (\mathsf{lam}^{\mathsf{M}}\, t^M)[\sigma^M]^{\mathsf{M}} = \mathsf{lam}\,(\mathsf{lam}\, t^M)[|\sigma^M|] \equiv \mathsf{lam}\,\big(\mathsf{lam}\,(t^M[|\sigma^M|\uparrow\uparrow])\big) = \mathsf{lam}^{\mathsf{M}}\,(t^M[\sigma^M\uparrow^{\mathsf{M}}]^{\mathsf{M}})
$$

8.6 Deriving the eliminator of a closed QIIT

In this subsection we show an example of the usefulness 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 constants for the eliminator but does not validate the existence of such constants. That is we do not derive the existence of inductive types from parametricity as done e.g. in [10] (assuming impredicativity). Our motivation was that we needed some systematic method like this to derive the eliminator for the syntax of type theory, which works for quotient inductive inductive types.

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.

$$(\Sigma(x : A).B)^{\mathsf{P}}\ (w : \Sigma(x : A).B) : \mathsf{U} = \Sigma(x^M : A^{\mathsf{P}}\ (\mathsf{proj}_1\ w)).B^{\mathsf{P}}\ (\mathsf{proj}_2\ w)$$
$$(a, b)^{\mathsf{P}} = (a^{\mathsf{P}}, b^{\mathsf{P}})$$
$$(\mathsf{proj}_1\ w)^{\mathsf{P}} = \mathsf{proj}_1\ w^{\mathsf{P}}$$
$$(\mathsf{proj}_2\ w)^{\mathsf{P}} = \mathsf{proj}_2\ w^{\mathsf{P}}$$

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

$$(a \equiv b)^{\mathsf{P}}\ (q : a \equiv b) : \mathsf{U} \qquad = a^{\mathsf{P}} \equiv^{q} b^{\mathsf{P}}$$
$$\mathsf{refl}^{\mathsf{P}} \qquad : a^{\mathsf{P}} \equiv^{\mathsf{refl}} a^{\mathsf{P}} = \mathsf{refl}$$

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.

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

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).

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

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.

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

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 **??**, $\mathbb{N}$ is closed while Vec isn't 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 **??**) and the interval I (section 2.2) 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 inductive type (we refer to the categorical notion of algebra where the constructors provide the data for the functor).

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.

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

We used this method to define the fields of record DModel in chapter **??**.

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 a context with three times as many elements Δ^{R}. 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 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 (f_{Ty} is given after applying a singleton contraction operation).

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

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 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.

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

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.

9 Extensions

In this section we show how to extend the type theory given in section 3 with more type formers and a universe. We do this by defining records which depend on elements of the previously defined records of the syntax.

9.1 Σ types

We extend our type theory given in section 3 with Σ types using the following definition. It depends obviously on the declaration of the theory and the core substitution calculus.

$$\begin{aligned}
&\mathsf{record}\ \mathsf{Sigma}\ (d : \mathsf{Decl})(c : \mathsf{Core}\, d)\\
&\quad \mathsf{open}\ \mathsf{Decl}\, d\\
&\quad \mathsf{open}\ \mathsf{Core}\, c\\
&\quad \Sigma \quad : (A : \mathsf{Ty}\,\Gamma) \to \mathsf{Ty}\,(\Gamma, A) \to \mathsf{Ty}\,\Gamma\\
&\quad \Sigma[] \quad : (\Sigma\, A\, B)[\sigma] \equiv \Sigma\,(A[\sigma])\,\big(B[\sigma \uparrow A]\big)\\
&\quad -,- \ : (a : \mathsf{Tm}\,\Gamma\, A) \to \mathsf{Tm}\,\Gamma\,(B[\langle a\rangle]) \to \mathsf{Tm}\,\Gamma\,(\Sigma\, A\, B)\\
&\quad \mathsf{proj}_1 : \mathsf{Tm}\,\Gamma\,(\Sigma\, A\, B) \to \mathsf{Tm}\,\Gamma\, A\\
&\quad \mathsf{proj}_2 : \big(a : \mathsf{Tm}\,\Gamma\,(\Sigma\, A\, B)\big) \to \mathsf{Tm}\,\Gamma\,\big(B[\langle \mathsf{proj}_1\, a\rangle]\big)\\
&\quad \Sigma\beta_1 \ : \mathsf{proj}_1\,(a, b) \equiv a\\
&\quad \Sigma\beta_2 \ : \mathsf{proj}_2\,(a, b) \equiv^{\Sigma\beta_1} b\\
&\quad \Sigma\eta \quad : (\mathsf{proj}_1\, w, \mathsf{proj}_2\, w) \equiv w\\
&\quad , [] \quad : (a, b)[\sigma] \equiv^{\Sigma[]} (a[\sigma], {}_{[][]\blacksquare\mathsf{ap}\,(B[-])\,\langle\rangle\circ\blacksquare[][]^{-1}} * b[\sigma])
\end{aligned}$$

In the last equation which gives the substitution law for pairing, we need to transport $b[\sigma]$ on the right hand side: given $a : \mathsf{Tm}\,\Theta\,A$ and $\sigma : \mathsf{Tms}\,\Gamma\,\Theta$, we have $b[\sigma] : \mathsf{Tm}\,\Gamma\,\big(B[\langle a\rangle][\sigma]\big)$, but we need a term of type $B[\sigma \uparrow A][\langle a[\sigma]\rangle]$. The equality that we need is given by $[][] \cdot \mathsf{ap}\,(B[-])\,\langle\rangle\circ \cdot [][]^{-1}$ where we use the following law which is easy to prove.

$$\langle\rangle\circ : \langle u\rangle \circ \sigma \equiv (\sigma \uparrow A) \circ \langle u[\sigma]\rangle$$

The definition of Σ types can be summarized as an isomorphism which is natural by $,[]$.

$$\Sigma\beta_1, \Sigma\beta_2 \circlearrowleft \qquad -,- \downarrow \quad \frac{a : \mathsf{Tm}\,\Gamma\,A \qquad \mathsf{Tm}\,\Gamma\,\big(B[\langle a\rangle]\big)}{\mathsf{Tm}\,\Gamma\,(\Sigma\,A\,B)} \quad \uparrow \mathsf{proj}_1, \mathsf{proj}_2 \qquad \circlearrowright \Sigma\eta$$

9.2 Natural numbers

The syntax for natural numbers is given as follows.

$$\begin{array}{ll}
\mathsf{record}\ \mathsf{Nat}\ (d : \mathsf{Decl})(c : \mathsf{Core}\,d) & \\
\quad \mathsf{open}\ \mathsf{Decl}\,d & \\
\quad \mathsf{open}\ \mathsf{Core}\,c & \\
\quad \mathbb{N} & : \mathsf{Ty}\,\Gamma \\
\quad \mathbb{N}[] & : \mathbb{N}[\sigma] \equiv \mathbb{N} \\
\quad \mathsf{zero} & : \mathsf{Tm}\,\Gamma\,\mathbb{N} \\
\quad \mathsf{suc} & : \mathsf{Tm}\,\Gamma\,\mathbb{N} \to \mathsf{Tm}\,\Gamma\,\mathbb{N} \\
\quad \mathsf{ind}_{\mathbb{N}} & : (P : \mathsf{Ty}\,(\Gamma, \mathbb{N})) \to \mathsf{Tm}\,\Gamma\,\big(P[\langle \mathsf{zero}\rangle]\big) \to \mathsf{Tm}\,(\Gamma, \mathbb{N}, P)\,\big(P[\mathsf{wk}, {}_{\mathbb{N}[]^{-1}}{*}(\mathsf{suc}\,{}_{\mathbb{N}[]}{*}\mathsf{vz})][\mathsf{wk}]\big) \\
 & \quad \to (n : \mathsf{Tm}\,\Gamma\,\mathbb{N}) \to \mathsf{Tm}\,\Gamma\,\big(P[\langle n\rangle]\big) \\
\quad \mathsf{zero}[] & : \mathsf{zero}[\sigma] \equiv^{\mathbb{N}[]} \mathsf{zero} \\
\quad \mathsf{suc}[] & : (\mathsf{suc}\,n)[\sigma] \equiv^{\mathbb{N}[]} \mathsf{suc}\,\big({}_{\mathbb{N}[]}{*}n[\sigma]\big) \\
\quad \mathsf{ind}_{\mathbb{N}}[] & : (\mathsf{ind}_{\mathbb{N}}\,P\,p_z\,p_s\,n)[\sigma] \equiv^{q_1} \mathsf{ind}_{\mathbb{N}}\,\big({}_{\mathbb{N}[]}{*}P[\sigma \uparrow \mathbb{N}]\big)\,\big({}_{q_2}{*}p_z[\sigma]\big)\,\big({}_{q_3}{*}p_s[\sigma \uparrow \mathbb{N} \uparrow P]\big)\,\big({}_{\mathbb{N}[]}{*}n[\sigma]\big) \\
\quad \mathbb{N}\beta_{\mathsf{zero}} & : \mathsf{ind}_{\mathbb{N}}\,P\,p_z\,p_s\,\mathsf{zero} \equiv p_z \\
\quad \mathbb{N}\beta_{\mathsf{suc}} & : \mathsf{ind}_{\mathbb{N}}\,P\,p_z\,p_s\,(\mathsf{suc}\,n) \equiv^{q_4} p_s[\mathsf{id}, {}_{\mathbb{N}[]^{-1}}{*}n, {}_{q_5}{*}\mathsf{ind}_{\mathbb{N}}\,P\,p_z\,p_s\,n]
\end{array}$$

Here q_1, q_2, q_3, q_4 can be given easily.

9.3 Universe

We have already given the synax for an empty universe (uninterpreted base type) in section 3.2. The following rules add the rules which say that this universe is closed under Π. In comparison to the rules given in section 3.3 we see that the difference is the decoration of everything with Els.

$$\begin{array}{ll}
\mathsf{record}\ \mathsf{FuncU}\ (d : \mathsf{Decl})(c : \mathsf{Core}\,d)(b : \mathsf{Base}\,d\,c) & \\
\quad \mathsf{open}\ \mathsf{Decl}\,d & \\
\quad \mathsf{open}\ \mathsf{Core}\,c & \\
\quad \mathsf{open}\ \mathsf{Base}\,b & \\
\quad \Pi & : (A : \mathsf{Tm}\,\Gamma\,\mathsf{U}) \to \mathsf{Tm}\,(\Gamma, \mathsf{El}\,A)\,\mathsf{U} \to \mathsf{Tm}\,\Gamma\,\mathsf{U} \\
\quad \Pi[] & : (\Pi\,A\,B)[\sigma] \equiv^{U[]} \Pi\,({}_{\mathsf{U}[]}{*}A[\sigma])\,({}_{\mathsf{El}[]}{*}B[\sigma \uparrow A]) \\
\quad \mathsf{lam} & : \mathsf{Tm}\,(\Gamma, \mathsf{El}\,A)\,(\mathsf{El}\,B) \to \mathsf{Tm}\,\Gamma\,(\mathsf{El}\,(\Pi\,A\,B)) \\
\quad \mathsf{app} & : \mathsf{Tm}\,\Gamma\,(\mathsf{El}\,(\Pi\,A\,B)) \to \mathsf{Tm}\,(\Gamma, \mathsf{El}\,A)\,(\mathsf{El}\,B) \\
\quad \Pi\beta & : \mathsf{app}\,(\mathsf{lam}\,t) \equiv t \\
\quad \Pi\eta & : \mathsf{lam}\,(\mathsf{app}\,t) \equiv t \\
\quad \mathsf{lam}[] & : (\mathsf{lam}\,t)[\sigma] \equiv^{\mathsf{El}[]\cdot\mathsf{ap}\,\mathsf{El}\,\Pi[]} \mathsf{lam}\,({}_{\mathsf{El}[]}{*}t[\sigma \uparrow A])
\end{array}$$

We can add other types into the universe analogously.

9.4 Identity type

The syntax for the identity type is given as follows.

$$
\begin{array}{l}
\mathsf{record}\ \mathsf{Iden}\ (d : \mathsf{Decl})(c : \mathsf{Core}\ d) \\
\quad \mathsf{open}\ \mathsf{Decl}\ d \\
\quad \mathsf{open}\ \mathsf{Core}\ c \\
\quad \mathsf{Id} \quad : (A : \mathsf{Ty}\,\Gamma) \to \mathsf{Tm}\,\Gamma\,A \to \mathsf{Tm}\,\Gamma\,A \to \mathsf{Ty}\,\Gamma \\
\quad \mathsf{Id}[] \quad : (\mathsf{Id}\,A\,a\,a')[\sigma] \equiv \mathsf{Id}\,A[\sigma]\,a[\sigma]\,a'[\sigma] \\
\quad \mathsf{refl} \quad : (a : \mathsf{Tm}\,\Gamma\,A) \to \mathsf{Id}\,A\,a\,a \\
\quad \mathsf{refl}[] \quad : (\mathsf{refl}\,a)[\sigma] \equiv \mathsf{refl}\,a[\sigma] \\
\quad \mathsf{ind}_{\mathsf{Id}} \quad : \Big(P : \mathsf{Ty}\,\big(\Gamma, A, A[\mathsf{wk}], \mathsf{Id}\,(A[\mathsf{wk}][\mathsf{wk}])\,(\mathsf{vs}\,\mathsf{vz})\,\mathsf{vz}\big)\Big)\,\big(p : \mathsf{Tm}\,(\Gamma, A)\,(P[\langle \mathsf{vz}\rangle, {}_{q_1*}\mathsf{refl}\,\mathsf{vz}])\big) \\
\qquad\qquad (a\,a' : \mathsf{Tm}\,\Gamma\,A)\,(r : \mathsf{Id}\,A\,a\,a') \to \mathsf{Tm}\,\Gamma\,\big(P[\langle u\rangle, {}_{q_2*}v, {}_{q_3*}r]\big) \\
\quad \mathsf{ind}_{\mathsf{Id}}[] : (\mathsf{ind}_{\mathsf{Id}}\,P\,p\,a\,a'\,q)[\sigma] \equiv^{q_4} \mathsf{ind}_{\mathsf{Id}}\,\big({}_{q_5*}P[\sigma \uparrow\uparrow\uparrow]\big)\,({}_{q_6*}p[\sigma \uparrow])\,(a[\sigma])\,(a'[\sigma])\,({}_{\mathsf{Id}[]*}r[\sigma]) \\
\quad \mathsf{Id}\beta \quad : \mathsf{ind}_{\mathsf{Id}}\,P\,p\,a\,a\,(\mathsf{refl}\,a)) \equiv^{q7} p[\langle a\rangle]
\end{array}
$$

We omit spelling out the $q_1, ..., q_7$ proofs.

10 Conclusions

We have for the first time presented a workable internal syntax of dependent type theory which only features typed objects. We have shown that the definition is feasible by constructing not only the standard model but also the logical predicate interpretation. Further interpretations are in preparation, e.g. the setoid interpretation and the presheaf interpretation. The setoid interpretation is essential for a formal justification of QITs and the presheaf interpretation is an essential ingredient to extend normalisation by evaluation [4] to dependent types. These constructions for dependent types require an attention to detail which can only convincingly demonstrated by a formal development. At the same time this approach would give us a certified implementation of key algorithms such as normalisation.

We would like to reflect our very general syntax for inductive-inductive types and QITs but this is a more serious challenge.

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: the use of presheaf models to justify guarded type theory has already been mentioned [27]. 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.

References

[1] The Agda Wiki, 2015. Available online.

[2] Thorsten Altenkirch. Extensional equality in intensional type theory. In *14th Symposium on Logic in Computer Science*, pages 412 – 420, 1999.

[3] Thorsten Altenkirch and James Chapman. Big-step normalisation. *Journal of Functional Programming*, 19(3-4):311–333, 2009.

[4] 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.

[5] Thorsten Altenkirch and Ambrus Kaposi. Normalisation by evaluation for dependent types. In *1st International Conference on Formal Structures for Computation and Deduction, FSCD 2016, June 22-26, 2016, Porto, Portugal*, pages 6:1–6:16, 2016.

[6] Thorsten Altenkirch and Ambrus Kaposi. Supplementary material for the paper Type Theory in Type Theory, 2016. Available online at the second author's website.

[7] Thorsten Altenkirch and Ambrus Kaposi. Type theory in type theory using quotient inductive types. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL 2016, pages 18–29, New York, NY, USA, 2016. ACM.

[8] 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 57–68, New York, NY, USA, 2007. ACM.

[9] Thorsten Altenkirch, Peter Morris, Fredrik Nordvall Forsberg, and Anton Setzer. A categorical semantics for inductive-inductive definitions. In *CALCO*, pages 70–84, 2011.

[10] 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.

[11] Jean-Philippe Bernardy, Patrik Jansson, and Ross Paterson. Proofs for free — parametricity for dependent types. *Journal of Functional Programming*, 22(02):107–152, 2012.

[12] 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.

[13] Ale Bizjak and Rasmus Ejlers Mgelberg. A model of guarded recursion with clock synchronisation. *Electronic Notes in Theoretical Computer Science*, 319:83 – 101, 2015.

[14] Matt Brown and Jens Palsberg. Self-representation in Girard's System U. *SIGPLAN Not.*, 50(1):471–484, January 2015.

[15] 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.

[16] Paolo Capriotti. Mutual and higher inductive types in homotopy type theory, 2014. Nottingham FP Lab Away Day.

[17] John Cartmell. Generalised algebraic theories and contextual categories. *Annals of Pure and Applied Logic*, 32:209–243, 1986.

[18] James Chapman. *Type checking and normalisation.* PhD thesis, University of Nottingham, 2008.

[19] James Chapman. Type theory should eat itself. *Electron. Notes Theor. Comput. Sci.*, 228:21–36, January 2009.

[20] Jesper Cockx and Andreas Abel. Sprinkles of extensionality for your vanilla type theory. In Silvia Ghilezan and Iveti Jelena, editors, *22nd International Conference on Types for Proofs and Programs, TYPES 2016*. University of Novi Sad, 2016.

[21] Nils Anders Danielsson. A formalisation of a dependently typed language as an inductive-recursive family. In Thorsten Altenkirch and Conor McBride, editors, *Types for Proofs and Programs*, volume 4502 of *Lecture Notes in Computer Science*, pages 93–109. Springer Berlin Heidelberg, 2007.

[22] 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.

[23] Peter Dybjer. Internal type theory. In *Types for Proofs and Programs*, pages 120–134. Springer, 1996.

[24] Robert Harper, Furio Honsell, and Gordon Plotkin. A framework for defining logics. *J. ACM*, 40(1):143–184, January 1993.

[25] M. Hofmann. *Extensional Concepts in Intensional Type Theory.* Thesis. University of Edinburgh, Department of Computer Science, 1995.

[26] Martin Hofmann. Syntax and semantics of dependent types. In *Extensional Constructs in Intensional Type Theory*, pages 13–54. Springer, 1997.

[27] 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.

[28] Ambrus Kaposi. *Type theory in a type theory with quotient inductive types.* PhD thesis, University of Nottingham, 2016.

[29] Nuo Li. *Quotient types in type theory.* Thesis. University of Nottingham, Department of Computer Science, 2015.

[30] Dan Licata. Running circles around (in) your proof assistant; or, quotients that compute, 2011. Available online.

[31] Peter LeFanu Lumsdaine and Mike Shulman. Semantics of higher inductive types, 2012. Note.

[32] Conor McBride. *Dependently Typed Functional Programs and their Proofs.* PhD thesis, University of Edinburgh, 1999.

[33] 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.

[34] N.P. Mendler. Quotient types via coequalizers in Martin-Löf type theory. In *Proceedings of the Logical Frameworks Workshop*, pages 349–361, 1990.

[35] Fredrik Nordvall Forsberg. *Inductive-inductive definitions.* PhD thesis, Swansea University, 2013.

[36] Ulf Norell. *Towards a practical programming language based on dependent type theory.* PhD thesis, Chalmers University of Technology, 2007.

[37] 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.

[38] 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.

[39] Gordon Plotkin and John Power. Notions of Computation Determine Monads. In Mogens Nielsen and Uffe Engberg, editors, *Foundations of Software Science and Computation Structures*, volume 2303 of *Lecture Notes in Computer Science*, pages 342–356. Springer Berlin Heidelberg, 2002.

[40] J. C. Reynolds. Types, abstraction and parametric polymorphism. In R. E. A. Mason, editor, *Information Processing 83, Proceedings of the IFIP 9th World Computer Congress*, pages 513–523. Elsevier Science Publishers B. V. (North-Holland), Amsterdam, 1983.

[41] Kristina Sojakova. Higher inductive types as homotopy-initial algebras. In *Proceedings of the 42Nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '15, pages 31–42, New York, NY, USA, 2015. ACM.

[42] Thomas Streicher. Investigations into intensional type theory. habilitation thesis, 1993. `http://www.mathematik.tu-darmstadt.de/~streicher/HabilStreicher.pdf`.

[43] The Univalent Foundations Program. Homotopy type theory: Univalent foundations of mathematics. Technical report, Institute for Advanced Study, 2013.

[44] Philip Wadler. Theorems for free! In *Functional Programming Languages and Computer Architecture*, pages 347–359. ACM Press, 1989.