Type Theory in Type Theory using Quotient Inductive Types

Thorsten Altenkirch Ambrus Kaposi
University of Nottingham
{txa,auk}@cs.nott.ac.uk

Abstract

We present an internal formalisation of dependent type theory 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 refering to preterms or a typability relation but defines directly well typed objects by an inductive definition. We use the elimination principle to define the set-theoretic and logical predicate interpretation. The work has been formalized using the Agda system extended with QITs using postulates.

Categories and Subject Descriptors D.3.1 [*Formal Definitions and Theory*]; F.4.1 [*Mathematical Logic*]: Lambda calculus and related systems

Keywords Higher Inductive Types, Homotopy Type Theory, Logical Relations, Metaprogramming

1. Introduction

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 (_@_) and abstraction (Λ). [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 2nd step with the intention that they should contain enough information so that we can reconstruct the typed objects.

[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.

```
data Ty   : Set where
  ι       : Ty
  _⇒_     : Ty → Ty → Ty
data Con  : Set where
  •       : Con
  _,_     : Con → Ty → Con
data Var  : Con → Ty → Set where
  zero    : Var (Γ , σ) σ
  suc     : Var Γ σ → Var (Γ , τ) σ
data Tm   : Con → Ty → Set where
  var     : Var Γ σ → Tm Γ σ
  _@_     : Tm Γ (σ ⇒ τ)
            → Tm Γ σ → Tm Γ τ
  Λ       : Tm (Γ , σ) τ → Tm Γ (σ ⇒ τ)
```

Figure 1. Simply typed λ-calculus

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:

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

(1) can be addressed by using inductive-inductive definitions [3] (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 were Higher Inductive Types 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 [25] 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 universe with explicit substitutions. We explain how a general recursor and eliminator can be derived. As a warm-up we define formally the *standard interpretation* which interprets every piece of syntax by the corresponding metatheoretic construction in section 4. Our main real-world example (section 5) is to formally give the logical predicate translation which has been introduced by Bernardy [4] to explain parametricity for dependent types. The constructions presented in this paper have been formalized in Agda, this is available as supplementary material.

1.2 Related Work

Our goal is to give a faithful exposition of the typed syntax of Type Theory in a first order setting in the sense that our syntax is finitary. Hence what we are doing is different from e.g. [7] which defines a higher order syntax to reflection. We also diverge from [19] which exploits the metatheoretic equality to make reflection feasible. We would like to have the freedom to interpret equality in any way we would like. We also note the work [11] which provides a very good motivation for our work but treats definitional equality in a non-standard way. Indeed, we want to express the syntax of type theory in a natural way without any special assumptions.

The work by Chapman [9] and the also earlier work by Danielsson [10] have a motivation very similar to ours. [10] uses implicit substitutions and this seems rather difficult to use in general, his definitions have rather adhoc character. [9] is more principled but relies on setoids and has to add a lot of boilerplate code to characterize families of setoids over a setoid explicitly. This boilerplate makes the definition in the end unusable in practice.

A nice application of type-theoretic metaprogramming is developed in [16] 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) [12, 15]. Indeed, one can summarize our work by saying that we construct an initial CwF using QITs. That something like this should be possible in principle was clear since a while, however, the progress reported here is that we have actually done it.

The style of the presentation can also be described as a generalized algebraic theory [8] which has been recently used by Coquand to give very concise presentations of type theory [5]. Our work shows that it should be possible to internalize this style of presentation in type theory itself.

2. The Type Theory we live in

We are working in a Martin-Löf Type Theory using Agda as a vehicle. That is our syntax is the one used by the functional programming language and interactive proof assistant Agda [21, 24]. In the paper, to improve readability we omit implicitly quantified variables whose types can be inferred from the context (in this respect we follow rather Idris [6]).

In Agda, Π types are written as $(\mathsf{x}{:}\mathsf{A}) \;\rightarrow\; \mathsf{B}$ for $\Pi(x : A).B$, implicit arguments are indicated by curly brackets $\{\mathsf{x}{:}\mathsf{A}\} \;\rightarrow\; \mathsf{B}$, these can be omitted if Agda can infer them from the types of later arguments. Agda supports mixfix notation, eg. function space can be denoted $\_\Rightarrow\_$ where the underscores show the placement of arguments. Underscores can also be used when defining a function if we are not interested in the value of the argument eg. the constant function can be defined as $\mathsf{const}\ \mathsf{x}\ \_\ =\ \mathsf{x}$. The keyword **data** is used to define inductively generated families and the keyword **record** is for defining dependent record types (iterated Σ-types) by listing their fields after the keyword **field**. Records are automatically modules (separate name spaces) in Agda, this is why they can be opened using the **open** keyword. In this case the field names become projection functions. Just as inductive types can have parameters, records and modules can also be parameterised (by a telescope of types), we use this feature of Agda to structure our code. Agda allows overloading of constructor names, we use this e.g. when using the same constructor $\_,\_$ for context extension and substitution extension. Equality (or the identity type) is denoted by $\_\equiv\_$ and has the constructor refl. We use the syntax $\mathsf{a} \equiv[\ \mathsf{p}\ ]\equiv \mathsf{b}$ to express that two objects in different types $\mathsf{a}{:}\mathsf{A}$ and $\mathsf{b}{:}\mathsf{B}$ are equal via an equality of their types $\mathsf{p}\ :\ \mathsf{A}\ \equiv\ \mathsf{B}$, see also section 3.

To support this flexible syntax Agda diverges from most programming languages in that space matters. E.g. $[\![\Gamma]\!]$ is just a variable name but $[\![\ \Gamma\ ]\!]$ is the application of the operation $[\![\_]\!]$ to Γ.

We use QITs which are not available in Agda, however we can simulate them by a technique pioneered by [18] for HITs. While QITs are a special case of HITs and are inspired by Homotopy Type Theory (HoTT), for most of the paper we shall work with a type theory with a strict equality, i.e. we assume that all equality proofs are equal. We will get back to this issue and explain how our work relates to Homotopy Type Theory in section 6.

However, we do 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* [2]. Recent work [5] 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. 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. As an example consider the recursor and the eliminator for the inductive type Ty defined in figure 1:

```
RecTy :   (Ty^M : Set)
          (ι^M : Ty^M)
          (⇒^M : Ty^M → Ty^M → Ty^M)
       → Ty → Ty^M
RecTy Ty^M ι^M ⇒^M ι       = ι^M
RecTy Ty^M ι^M ⇒^M (A ⇒ B)
   = ⇒^M (RecTy Ty^M ι^M ⇒^M A)
         (RecTy Ty^M ι^M ⇒^M B)
ElimTy    :  (Ty^M : Ty → Set)
             (ι^M   : Ty^M ι)
             (⇒^M  :   {A : Ty} (A^M : Ty^M A)
                       {B : Ty} (B^M : Ty^M B)
                    → Ty^M (A ⇒ B))
          → (A : Ty) → Ty^M A
ElimTy Ty^M ι^M ⇒^M ι = ι^M
ElimTy Ty^M ι^M ⇒^M (A ⇒ B)
   =  ⇒^M (ElimTy Ty^M ι^M ⇒^M A)
          (ElimTy Ty^M ι^M ⇒^M B)
```

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 — however they can be reduced to a single inductive family by using an extra parameter of type Bool which provides the information which type is meant.

However, there are examples of mutual definitions which are not covered by this explanation: that is if we define a type and a family which depends on this type mutually. In this case we may also refer to constructors which have been defined previously. A canonical example for this is a fragment of the definition of the syntax of dependent types where we only define types and contexts (figure 2). Indeed we have to define types and contexts at the same

```
data Con : Set
data Ty  : Con → Set
data Con where
  •      : Con
  _,_  : (Γ : Con) → Ty Γ → Con
data Ty where
  U      : ∀ {Γ} → Ty Γ
  Π      : ∀ {Γ} (A : Ty Γ) (B : Ty (Γ , A)) → Ty Γ
```

Figure 2. An example of an inductive-inductive type: contexts and types in a dependent type theory

time but types are indexed by contexts to express that we want to define the type of valid types in a given context. The constructor _,_ appears in the type of the Π-constructor to express that the codomain type lives in the context extended by the domain type.

The definition of such *inductive-inductive* types in Agda is standard. We first declare the types of all the types we want to define inductively but without giving the constructors and then complete

```
module RecConTy where
  record Motives : Set₁ where
    field
      Con^M  : Set
      Ty^M   : Con^M → Set
  record Methods (M : Motives) : Set₁ where
    open Motives M
    field
      •^M      : Con^M
      _,C^M_ : (Γ^M : Con^M) → Ty^M Γ^M → Con^M
      U^M      : {Γ^M : Con^M} → Ty^M Γ^M
      Π^M      : {Γ^M : Con^M} (A^M : Ty^M Γ^M)
                 (B^M : Ty^M (Γ^M ,C^M A^M)) → Ty^M Γ^M
  module rec (M : Motives) (m : Methods M) where
    open Motives M
    open Methods m
    RecCon : Con → Con^M
    RecTy : {Γ : Con} (A : Ty Γ) → Ty^M (RecCon Γ)
    RecCon •          = •^M
    RecCon (Γ , A)  = RecCon Γ ,C^M RecTy A
    RecTy  U          = U^M
    RecTy  (Π A B)  = Π^M (RecTy A) (RecTy B)
```

Figure 3. Recursor for the inductive-inductive type of figure 2

the definition of the constructors when all the type signatures are in scope.

As before, programming with inductive-inductive types can be reduced to using the elimination constants - see figures 3 and 4 for those of the mutual inductive types Con and Ty defined in figure 2. To make working with complex elimination constants feasible we organize their parameters into records: Motives, Methods. The motives and methods for the recursor can be defined just by adding $^\mathsf{M}$ indices to the types of the types and constructors. Defining the motives and methods for the eliminator is more subtle: we need to define families over the types and fibers of those families over the constructors taking into account the mutual dependencies — see section 5.3 for a generic way of deriving these. The β rules can be added to th system by pattern matching on the constructors — these rules are the same for the recursor and the eliminator.

The categorical semantics of inductive-inductive types has been explored in [3, 20]. From a computational point of view inductive-inductive types are unproblematic and pattern matching can be reduced to using elimination constants which are derived from the type signature and constructor types.

2.2 Quotient Inductive Types (QITs)

One of the main applications of Higher Inductive Types in Homotopy Type Theory is to represent types with non-trivial equalities corresponding to the path spaces of topological spaces. Here we are working in a Type Theory with a strict equality, i.e. all higher path spaces are trivial. However, there are still interesting applications for these degenerate HITs which we call Quotient Inductive Types (QITs). E.g. in [25] QITs (even though not by that name) are used to define the constructible hierarchy of sets in an encoding of set theory within HoTT and later to define the Cauchy Reals. What is striking is that in both cases ordinary quotient types would not have been sufficient but would have required some form of the axiom of choice.

```
module ElimConTy where
  record Motives : Set₁ where
    field
      Conᴹ  :  Con → Set
      Tyᴹ   :  {Γ : Con} → Conᴹ Γ → Ty Γ → Set
  record Methods (M : Motives) : Set₁ where
    open Motives M
    field
      •ᴹ    :  Conᴹ •
      _,Cᴹ_ :  {Γ : Con} (Γᴹ : Conᴹ Γ)
               {A : Ty Γ} (Aᴹ : Tyᴹ Γᴹ A)
            →  Conᴹ (Γ , A)
      Uᴹ    :  {Γ : Con} {Γᴹ : Conᴹ Γ}
            →  Tyᴹ Γᴹ U
      Πᴹ    :  {Γ : Con} {Γᴹ : Conᴹ Γ}
               {A : Ty Γ} (Aᴹ : Tyᴹ Γᴹ A)
               {B : Ty (Γ , A)}
               (Bᴹ : Tyᴹ (Γᴹ ,Cᴹ Aᴹ) B)
            →  Tyᴹ Γᴹ (Π A B)
  module elim (M : Motives) (m : Methods M) where
    open Motives M
    open Methods m
    ElimCon : (Γ : Con) → Conᴹ Γ
    ElimTy :  {Γ : Con} (A : Ty Γ)
           → Tyᴹ (ElimCon Γ) A
    ElimCon •          =  •ᴹ
    ElimCon (Γ , A)  =  ElimCon Γ ,Cᴹ ElimTy A
    ElimTy  U          =  Uᴹ
    ElimTy  (Π A B)  =  Πᴹ (ElimTy A) (ElimTy B)
```

Figure 4. Eliminator for the inductive-inductive type of figure 2

While this is not the use of quotient inductive types which is directly relevant for our representation of dependently typed calculi it is worthwhile to explain this in some detail to make clear what QITs are about.

Our goal is to define infinitely branching trees where the actual order of subtrees doesn't matter. We start by defining the type of infinite trees:

```
data T₀ : Set where
  leaf : T₀
  node : (ℕ → T₀) → T₀
```

and now we specify an equivalence relation which allows us to use an isomorphism to reorder a tree locally.

```
data _~_ :  T₀ → T₀ → Set where
  leaf   :  leaf ∼ leaf
  node   :  {f g : ℕ → T₀} → (∀ {n} → f n ∼ g n)
         →  node f ∼ node g
  perm   :  (g : ℕ → T₀) (f : ℕ → ℕ) → isIso f
         →  node g ∼ node (g ∘ f)
```

Here isIso : (ℕ → ℕ) → Set specifies that the argument is an isomorphism, i.e. there is an inverse function. Quotient types [14] can be postulated as shown in figure 5. With the help of this, we can construct the type:

```
data _/_ (A : Set) (R : A → A → Set) : Set where
  [_] : A → A / R
postulate
  [_]≡  :  ∀ {A} {R : A → A → Set} {a b : A}
        → R a b → [ a ] ≡ [ b ]
module Elim_/_
  (A : Set) (R : A → A → Set)
  (Qᴹ : A / R → Set)
  ([_]ᴹ : (a : A) → Qᴹ [ a ])
  ([_]≡ᴹ  :  {a b : A} (r : R a b)
          → [ a ]ᴹ ≡[ ap Qᴹ [ r ]≡ ]≡ [ b ]ᴹ)
  where
    Elim : (x : A / R) → Qᴹ x
    Elim [ x ] = [ x ]ᴹ
```

Figure 5. The constructors and elimination principle for quotient types in HoTT-style. Note that the [_]≡ equality constructor needs to be postulated and pattern matching on elements of A / R is disallowed except when defining Elim (this is not checked by Agda, we need to ensure this by hand): the only way to define a function from A / R should be using the eliminator.

```
T : Set
T = T₀ / _~_
```

Note that this doesn't require to show that _~_ is an equivalence relation but the resulting type is equivalent to the quotient with the equivalence closure of the given relation. The elements of T are equivalence classes [t] : $T_0 \to T_0$ / _~_ and given p : t ∼ t' we have [p]≡ : [t] ≡ [t']. The elimination principle Elim allows us to lift a function [_]ᴹ which respects _~_ (expressed by [_]≡ᴹ) to any element of the quotient.

We would expect that we should be able to lift node to equivalence classes, i.e. we would like to define a function node' : (ℕ → T) → T such that [node f] ≡ node' ([_] ∘ f). However, it seems not possible to do this. To see what the problem is it is instructive to solve the same exercise for finitely branching trees. It turns out that we have to sequentially iterate the eliminator depending on the branching factor. However, clearly this approach doesn't work for infinite trees. And indeed, in general assuming that we can lift function types of equivalence classes is equivalent to the axiom of choice. And this is an intuitionistic taboo since it entails the excluded middle.

However, if we use a QIT and specify the constructor for equality at the same time we avoid this issue altogether.

```
data T :  Set where
  leaf :  T
  node :  (ℕ → T) → T
  perm :  (g : ℕ → T) (f : ℕ → ℕ) → isIso f
       → node g ≡ node (g ∘ f)
```

The main difference here is that we specify the new equality and the constructors at the same time. We also do not need to assume that the relation is a congruence because this is provable in general for the equality type. The dependent eliminator is given in figure 6: we see that we also need to interpret the equation when we eliminate from T (pattern matching on T should be avoided after defining the eliminator).

```
module ElimT
  (T^M     :   T → Set)
  (leaf^M  :   T^M leaf)
  (node^M  :   {f : ℕ → T} (f^M : (n : ℕ) → T^M (f n))
           →  T^M (node f))
  (perm^M  :   {g : ℕ → T} (g^M : (n : ℕ) → T^M (g n))
               (f : ℕ → ℕ) (p : isIso f)
           →  node^M g^M ≡[ ap T^M (perm g f p) ]≡
               node^M (g^M ∘ f))
  where
  Elim : (t : T) → T^M t
  Elim leaf = leaf^M
  Elim (node f) = node^M (λ n → Elim (f n))
```

Figure 6. The eliminator for the quotient-inductive type T

```
data Con  : Set
data Ty   : Con → Set
data Tms  : Con → Con → Set
data Tm   : ∀ Γ → Ty Γ → Set
```

Figure 7. Signature of the syntax

```
data Con where
  •       : Con
  _,_     : Con → Ty → Con
data Ty where
  _[_]T  : Ty Δ → Tms Γ Δ → Ty Γ
  Π       : (A : Ty Γ) (B : Ty (Γ , A)) → Ty Γ
  U       : Ty Γ
  El      : (A : Tm Γ U) → Ty Γ
```

Figure 8. Constructors for contexts and types

3. Representing the syntax of Type Theory

We are now going to apply the tools introduced in the previous section to formalize a simple dependent type theory, that is a type theory with Π-types and an uninterpreted family denoted by $U : Type$ and $El : U \to Type$. Despite the naming this is not a universe, but a base type in the same way as ι is a base type for the simply typed λ calculus. Without this the syntax would be trivial as there would be no types or terms we could construct.

The signature of the QIT we are using to represent the syntax of type theory is given in figure 7. We have already seen contexts Con and Types Ty in our previous example for an inductive-inductive definition. We extend this by explicit substitutions (Tms) which are indexed by two contexts and terms Tm which are indexed by a context and a type in the given context.

The constructors for contexts are exactly the same as in the previous example but we are introducing some new constructors for types (figure 8). Most notably we introduce a substitution operator _[_]T which applies a substitution from Γ to Δ to a type in context Δ producing a type in context Γ. The contravariant character of substitution arises from the desire to semantically interpret substitutions as functions going in the same direction, as we will see in detail in section 4. Syntactically it is good to think of an element of Tms Γ Δ as a sequence of terms in context Γ which inhabit all the types in Δ. This intuition is reflected in the syntax for substitutions (figure 9): ϵ is the empty sequence of terms, and _,_ extends a given sequence by an additional term. It is worthwhile to note that while the type which is added lives in the previous target context Δ, the term has free variables in Γ which makes it necessary to apply the substitution operator on types: A [δ]T. We also introduce inverses to _,_, i.e. projections. The first one π_1 produces a substitution by forgetting the last term. Since we have explicit substitutions we also have explicit composition _∘_ of substitutions and consequently also the identity substitution which will be essential when we reconstruct variables.

```
data Tms where
  ϵ     :   Tms Γ •
  _,_   :   (δ : Tms Γ Δ) → Tm Γ (A [ δ ]T)
        →  Tms Γ (Δ , A)
  id    :   Tms Γ Γ
  _∘_   :   Tms Δ Σ → Tms Γ Δ → Tms Γ Σ
  π₁    :   Tms Γ (Δ , A) → Tms Γ Δ
```

Figure 9. Constructors for substitutions

```
data Tm where
  _[_]t :   Tm Δ A → (δ : Tms Γ Δ)
         →  Tm Γ (A [ δ ]T)
  π₂     :   (δ : Tms Γ (Δ , A)) → Tm Γ (A [ π₁ δ ]T)
  app    :   Tm Γ (Π A B) → Tm (Γ , A) B
  lam    :   Tm (Γ , A) B → Tm Γ (Π A B)
```

Figure 10. Constructors for terms

For terms (see figure 10) we also have a contravariant substitution operator _[_]t whose type uses the substitution operator for types. We also introduce the second projection π_2 which projects out the final term from a non-empty substitution. Finally, we introduce app and lam which construct an isomorphism between Tm (Γ , A) B and Tm Γ (Π A B).

Let's explore how our categorically inspired syntax can be used to derive more mundane syntactical components such as variables. First we derive the weakening substitution:

```
wk : ∀ {Γ} {A : Ty Γ} → Tms (Γ , A) Γ
wk = π₁ id
```

We need to derive typed de Bruijn variables as in the example for the simply typed λ-calculus. However, we have to be more precise because the result types live in the extended context and hence we need weakening. However, the definitions are straightforward:

```
vz : Tm (Γ , A) (A [ wk ]T)
vz = π₂ id
vs : Tm Γ A → Tm (Γ , B) (A [ wk ]T)
vs x = x [ wk ]t
```

Now we turn our attention to the constructors giving the equations. To define these we sometimes need to *transport* elements along equalities. To simplify this we introduce a number of convenient operations. coe turns an equality between types into a function:

```
coe : A ≡ B → A → B
```

Specifically in our current construction we often want to coerce terms along an equality of syntactic types, to facilitate this we

```
 [id]T    : A [ id ]T ≡ A
 [][]T    : A [ δ ]T [ σ ]T ≡ A [ δ ∘ σ ]T
 U[]      : U [ δ ]T ≡ U
 El[]     : El A [ δ ]T ≡ El (coe (TmΓ≡ U[]) (A [ δ ]t))
_ ↑ _ :   (δ : Tms Γ Δ) (A : Ty Δ)
       → Tms (Γ , A [ δ ]T) (Δ , A)
δ ↑ A = (δ ∘ π₁ id) , coe (TmΓ≡ [][]T) (π₂ id)
 Π[]      : (Π A B) [ δ ]T ≡ Π (A [ δ ]T) (B [ δ ↑ A ]T)
```

Figure 11. Equations for types

```
idl   : id ∘ δ ≡ δ
idr   : δ ∘ id ≡ δ
ass   : (σ ∘ δ) ∘ ν ≡ σ ∘ (δ ∘ ν)
,∘    : (δ , t) ∘ σ ≡ (δ ∘ σ) , coe (TmΓ≡ [][]T) (t [ σ ]t)
π₁β   : π₁ (δ , t) ≡ δ
πη    : (π₁ δ , π₂ δ) ≡ δ
εη    : {σ : Tms Γ •} → σ ≡ ε
```

Figure 12. Equations for substitutions

introduce an operation which lifts an equality between syntactic types to an equality of semantic types of terms:

```
TmΓ≡ :   {A₀ : Ty Γ} {A₁ : Ty Γ} (A₂ : A₀ ≡ A₁)
       → Tm Γ A₀ ≡ Tm Γ A₁
```

A more general version of this function is ap (apply path in HoTT terminology):

```
ap :   (f : A → B) {a₀ a₁ : A} (a₂ : a₀ ≡ a₁)
     → f a₀ ≡ f a₁
```

We introduce no additional equations on contexts - however, the equality of contexts is not syntactic since they contain types which has non-trivial equalities.

Let us turn our attention to syntactic types Ty, see figure 11. The first two equations explain how substitution interacts with the identity substitution and composition. The remaining equations explain how substitutions move inside the other type constructors. When introducing the equation for Π we notice that we need to lift a substitution along a type, that is given an element of Tms Γ Δ we want to derive Tms (Γ , A [δ]T) (Δ , A). This is accomplished by _ ↑ _ which can be defined using existing constructors of substitutions and terms, namely δ ↑ A = (δ ∘ wk , vz). However this doesn't type check since the second component of the substitution should have type Tm (Γ , A [δ]T) (A [δ ∘ wk]T) but vz has type Tm (Γ , A[δ]T) (A[δ]T [wk]T). However using the equation just introduced which describes the interaction between substitution and composition can be used to fix this issue.

The equations for substitutions (figure 12) state that substitutions form a category and how composition commutes with _,_ which relies again on [][]T. We also state that π_1 works as expected and that surjective pairing holds. There is only one substitution into the empty context ($\epsilon\eta$), this entails that ϵ is a terminal object in the category of substitutions.

The equations for terms (figure 13) start similarly to those for types: first we explain how term substitution interacts with the identity substitution and composition. Unsurprisingly, these laws are *upto* the corresponding laws for types. We state the law for the second projection whose typing relies on the equation for the first projection. The equations $\Pi\beta$ and $\Pi\eta$ state that lam and app are inverse to each other. lam[] explains how substitutions can be moved into λ-abstractions. This law refers to the substitution law for Π-types which can be viewed as an example of the Beck-Chevalley condition. Note that a corresponding law for app is derivable:

```
[id]t : t [ id ]t ≡[ TmΓ≡ [id]T ]≡ t
[][]t : (t [ δ ]t) [ σ ]t ≡[ TmΓ≡ [][]T ]≡ t [ δ ∘ σ ]t
π₂β  : π₂ (δ , a) ≡[ TmΓ≡ (ap (_[_]T A) π₁β) ]≡ a
Πβ   : app (lam t) ≡ t
Πη   : lam (app t) ≡ t
lam[] : (lam t) [ δ ]t ≡[ TmΓ≡ Π[] ]≡ lam (t [ δ ↑ A ]t)
```

Figure 13. Equations for terms

```
app (coe (TmΓ≡ Π[]) (t [ δ ]t))
   ≡ ⟨ ap (λ z → app (coe (TmΓ≡ Π[]) (z [ δ ]t)))
                         (Πη ⁻¹) ⟩
app (coe (TmΓ≡ Π[]) ((lam (app t)) [ δ ]t))
   ≡ ⟨ ap app lam[] ⟩
app (lam (app t [ δ ↑ A ]t))
   ≡ ⟨ Πβ ⟩
app t [ δ ↑ A ]t
```

The equations can be summarized as follows: the contexts form a category with a terminal object and we have the corresponding laws [id]T/t, [][]T/t for substitutions of types and terms; we have substitution rules for type formers U[], El[], Π[]; the rest of the equations express two natural isomorphisms, one for the substitution extension _,_ and one for Π: the β laws express that going down and up is the identity, the η laws express that going up and then down is the identity, while naturalities give the relationship with substitutions (an isomorphism is natural in both directions if it is natural in one direction).

$$\_,\_ \downarrow \;\; \frac{\rho : \mathsf{Tms}\,\Gamma\,\Delta \qquad \mathsf{Tm}\,\Gamma\,(\mathsf{A}\,[\,\rho\,]\mathsf{T})}{\mathsf{Tms}\,\Gamma\,(\Delta\,,\,\mathsf{A})} \;\; \uparrow \pi_1\,,\,\pi_2$$

$$\mathsf{lam} \downarrow \;\; \frac{\mathsf{Tm}\,(\Gamma\,,\,\mathsf{A})\,\mathsf{B}}{\mathsf{Tm}\,\Gamma\,(\Pi\,\mathsf{A}\,\mathsf{B})} \;\; \uparrow \mathsf{app}$$

We show how a more conventional application operator can be derived. First we introduce one term substitution:

```
<_> : Tm Γ A → Tms Γ (Γ , A)
< t > = id , coe (TmΓ≡ ([id]T ⁻¹)) t
```

Note that here we need to apply the equation for identity substitutions backward exploiting symmetry for equations _$^{-1}$. Given this it is easy to state and derive ordinary application:

```
_ $ _ :   Tm Γ (Π A B) → (u : Tm Γ A)
        → Tm Γ (B [ < u > ]T)
t $ u = (app t) [ < u > ]t
```

We prefer to use the categorical combinators in the definition of the syntax since they are easier to work with, avoiding unnecessary introduction of single term substitutions which correspond to using id.

We define the recursor and the eliminator analogously to the examples in section 2. The motives and methods have the same names as the types and constructors with an added $^{\mathsf{M}}$ index. We list the motives and the methods for types for the eliminator in figure 14. Note the usage of lifted congruence rules such as TyΓ$^{\mathsf{M}}$≡ and TmΓ$^{\mathsf{M}}$≡ and how lifting the coerces is done. Also we define a lifting of the _ ↑ _ helper function.

```
record Motives : Set₁ where
  field
    Conᴹ    :  Con → Set
    Tyᴹ     :  Conᴹ Γ → Ty Γ → Set
    Tmsᴹ    :  Conᴹ Γ → Conᴹ Δ → Tms Γ Δ
               → Set
    Tmᴹ     :  Tyᴹ Γᴹ A → Tm Γ A → Set
record Methods (M : Motives) : Set₁ where
  open Motives M
  field
    ...
    _[_]Tᴹ :  (δᴹ : Tmsᴹ Γᴹ Δᴹ δ)
           → Tyᴹ Γᴹ (A [ δ ]T)
    Uᴹ     :  {Γᴹ : Conᴹ Γ} → Tyᴹ Γᴹ U
    Elᴹ    :  (Âᴹ : Tmᴹ Γᴹ Uᴹ Â) → Tyᴹ Γᴹ (El Â)
    Πᴹ     :  (Aᴹ : Tyᴹ Γᴹ A)
              (Bᴹ : Tyᴹ (Γᴹ ,Cᴹ Aᴹ) B)
           → Tyᴹ Γᴹ (Π A B)
    ...
    [id]Tᴹ :  Aᴹ [ idᴹ ]Tᴹ ≡[ TyΓᴹ≡ [id]T ]≡ Aᴹ
    [][]Tᴹ :  Aᴹ [ δᴹ ]Tᴹ [ σᴹ ]Tᴹ
              ≡[ TyΓᴹ≡ [][]T ]≡
              Aᴹ [ δᴹ ∘ᴹ σᴹ ]Tᴹ
    U[]ᴹ   :  Uᴹ [ δᴹ ]Tᴹ ≡[ TyΓᴹ≡ U[] ]≡ Uᴹ
    El[]ᴹ  :  Elᴹ Âᴹ [ δᴹ ]Tᴹ
              ≡[ TyΓᴹ≡ El[] ]≡
              Elᴹ (coe (TmΓᴹ≡ U[]ᴹ refl)
                       (Âᴹ [ δᴹ ]tᴹ))
  _↑ᴹ_ :  (δᴹ : Tmsᴹ Γᴹ Δᴹ δ) (Aᴹ : Tyᴹ Δᴹ A)
       → Tmsᴹ (Γᴹ ,Cᴹ Aᴹ [ δᴹ ]Tᴹ)
              (Δᴹ ,Cᴹ Aᴹ) (δ ↑ A)
              (δ ↑ A)
  δᴹ ↑ᴹ Aᴹ =  (δᴹ ∘ᴹ π₁ᴹ idᴹ)
           ,sᴹ  coe (TmΓᴹ≡ [][]Tᴹ refl) (π₂ᴹ idᴹ)
  field
    Π[]ᴹ :  Πᴹ Aᴹ Bᴹ [ δᴹ ]Tᴹ ≡[ TyΓᴹ≡ Π[] ]≡
            Πᴹ (Aᴹ [ δᴹ ]Tᴹ) (Bᴹ [ δᴹ ↑ᴹ Aᴹ ]Tᴹ)
    ...
```

Figure 14. The motives and some methods for the eliminator for the syntax

```
M : Motives
M = record
  { Conᴹ                =  Set
  ; Tyᴹ   ⟦Γ⟧           =  ⟦Γ⟧ → Set
  ; Tmsᴹ  ⟦Γ⟧ ⟦Δ⟧       =  ⟦Γ⟧ → ⟦Δ⟧
  ; Tmᴹ   ⟦Γ⟧ ⟦A⟧       =  (γ : ⟦Γ⟧) → (⟦A⟧ γ)
  }
⟦_⟧C : Con            → Set
⟦_⟧T : Ty Γ           → ⟦ Γ ⟧C → Set
⟦_⟧s : Tms Γ Δ        → ⟦ Γ ⟧C → ⟦ Δ ⟧C
⟦_⟧t : (t : Tm Γ A)   → (γ : ⟦ Γ ⟧C) → ⟦ A ⟧T γ
```

Figure 15. Motives for the standard model and how we would define the type of the interpretation functions using usual Agda syntax

```
m : Methods M
m = record
  { •ᴹ                    =  ⊤
  ; ⟦Γ⟧ ,Cᴹ ⟦A⟧           =  Σ ⟦Γ⟧ ⟦A⟧
  ; ⟦A⟧ [ ⟦δ⟧ ]Tᴹ  γ      =  ⟦A⟧ (⟦δ⟧ γ)
  ; Uᴹ             _      =  ⟦U⟧
  ; Elᴹ ⟦t⟧        γ      =  ⟦El⟧ (⟦t⟧ γ)
  ; Πᴹ ⟦A⟧ ⟦B⟧     γ      =  (x : ⟦A⟧ γ) → ⟦B⟧ (γ , x)
  ; ϵᴹ             _      =  tt
  ; ⟦δ⟧ ,sᴹ ⟦t⟧    γ      =  ⟦δ⟧ γ , ⟦t⟧ γ
  ; idᴹ            γ      =  γ
  ; ⟦δ⟧ ∘ᴹ ⟦σ⟧     γ      =  ⟦δ⟧ (⟦σ⟧ γ)
  ; π₁ᴹ ⟦δ⟧        γ      =  proj₀ (⟦δ⟧ γ)
  ; ⟦t⟧ [ ⟦δ⟧ ]tᴹ  γ      =  ⟦t⟧ (⟦δ⟧ γ)
  ; π₂ᴹ ⟦δ⟧        γ      =  proj₁ (⟦δ⟧ γ)
  ; appᴹ ⟦t⟧       γ      =  ⟦t⟧ (proj₀ γ) (proj₁ γ)
  ; lamᴹ ⟦t⟧       γ      =  λ a → ⟦t⟧ (γ , a)
    ...
  }
```

Figure 16. Methods for the standard model

An interpretation of the syntax can be given by providing elements of the records Motives and Methods. Soundness is ensured by the methods for equality constructors. This way a model of Type Theory can be viewed as an algebra of the syntax.

4. The standard model

As a first sanity check of our syntax we define the standard model where 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. We use the recursor to define this interpretation, the motives are given in figure 15. We use an idealized record notation where fields can have parameters (in real Agda these need to be lambda expressions). The parameters of the fields of the motives are the results of the recursive calls of the recursor.

The definition of the methods is now straightforward (figure 16). In particular the interpretation of substitution nicely explains the contravariant character of the substitution rules. Note that $[\![\mathsf{U}]\!] : \mathsf{Set}$ and $[\![\mathsf{El}]\!] : [\![\mathsf{U}]\!] \to \mathsf{Set}$ are module parameters.

We have omitted the interpretation of all the equational constants because they are trivial: all of them are refl because the two sides are actually convertible in the metatheory.

A consequence of the standard model is soundness, that is in our case we can show that there is no closed term of U because we can instantiate U with the empty type. It should also be clear that to construct the standard model we need a stronger metatheory than the object theory we are considering. In our case this is given by the presence of an additional universe (here we have to eliminate over Set_1).

5. The logical predicate interpretation

In this section, after briefly introducing parametricity for a simple dependent type theory we describe our formalisation of this interpretation using the syntax given in section 3. This is a real-world example of the usefulness of our representation of the syntax; note that not only the domain of our interpretation but also the codomain is the syntax.

5.1 Logical relations for dependent types

Logical relations were introduced in computer science by Reynolds [23] for expressing the idea of representation-independence in the context of the polymorphic λ-calculus. Reynold's abstraction theorem (also called parametricity) states that logically related interpretations of a term are logically related in the relation generated by the type of the term. This was later extended to dependent types by [4]: 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.

We describe the parametric interpretation for an explicit substitution calculus with named variables, universes a la Russel, Pi types and one universe. The syntax of contexts, terms and types is the following:

$$\begin{array}{lcl}
\Gamma & ::= & \bullet \mid \Gamma\,,\mathsf{x} : \mathsf{A} \\
\mathsf{t,u,A,B} & ::= & \mathsf{x} \mid \mathsf{Set} \mid (\mathsf{x{:}A}) \to \mathsf{B} \mid \lambda\,\mathsf{x} \to \mathsf{t} \mid \mathsf{t\,u} \\
 & & \mid \mathsf{t}\,[\,\sigma\,] \\
\sigma\,,\delta & ::= & \epsilon \mid (\sigma\,,\mathsf{x} \mapsto \mathsf{t}) \mid \sigma \circ \delta \mid \mathsf{id}
\end{array}$$

$\mathsf{t}\,[\,\sigma\,]$ is the notation for a substituted term, weakening is implicit. we omit the typing rules, they are standard and are given in the formal development.

We define the unary logical predicate operation $\_^{\mathsf{P}}$ on the syntax following [4]. 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, terms (which here include types) and substitutions:

$$\frac{\Gamma\ \text{valid}}{\Gamma^{\mathsf{P}}\ \text{valid}} \qquad \frac{\Gamma \vdash \mathsf{t} : \mathsf{A}}{\Gamma^{\mathsf{P}} \vdash \mathsf{t}^{\mathsf{P}} : (\mathsf{A}^{\mathsf{P}})\,\mathsf{t}} \qquad \frac{\sigma : \Gamma \longrightarrow \Delta}{\sigma^{\mathsf{P}} : \Gamma^{\mathsf{P}} \longrightarrow \Delta^{\mathsf{P}}}$$

The second rule expresses parametricity, i.e. the internal fundamental theorem for logical relations. The rule for terms, when specialised to elements of the universe expresses that A^{P} is a predicate over A:

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

The operation $\_^{\mathsf{P}}$ is defined by induction on the syntax as follows:

$$\begin{array}{lcl}
\bullet^{\mathsf{P}} & = & \bullet \\
(\Gamma\,,\mathsf{x} : \mathsf{A})^{\mathsf{P}} & = & \Gamma^{\mathsf{P}}\,,\mathsf{x} : \mathsf{A}\,,\mathsf{x}^{\mathsf{M}} : \mathsf{A}^{\mathsf{P}}\,\mathsf{x}
\end{array}$$

```
record Conᵐ (Γ : Con) : Set where
  field
    =C : Con
    Pr : Tms =C Γ
Tyᵐ : Conᵐ Γ → Ty Γ → Set
Tyᵐ Γᴹ A = Ty (=C Γᴹ , A [ Pr Γᴹ ]T)
record Tmsᵐ (Γᴹ : Conᵐ Γ) (Δᴹ : Conᵐ Δ)
            (ρ : Tms Γ Δ) : Set where
  field
    =s : Tms (=C Γᴹ) (=C Δᴹ)
    PrNat : (Pr Δᴹ) ∘ =s ≡ ρ ∘ (Pr Γᴹ)
Tmᵐ : (Γᴹ : Conᵐ Γ) → Tyᵐ Γᴹ A → Tm Γ A → Set
Tmᵐ Γᴹ Aᴹ a = Tm (=C Γᴹ) (Aᴹ [ < a [ Pr Γᴹ ]t > ]T)
M : Motives
M = record { Conᴹ  = Conᵐ
           ; Tyᴹ   = Tyᵐ
           ; Tmsᴹ  = Tmsᵐ
           ; Tmᴹ   = Tmᵐ }
```

Figure 17. Motives for the eliminator in the logical predicate interpretation

$$\begin{array}{lcl}
\mathsf{x}^{\mathsf{P}} & = & \mathsf{x}^{\mathsf{M}} \\
\mathsf{Set}^{\mathsf{P}} & = & \lambda\,\mathsf{A} \to (\mathsf{A} \to \mathsf{Set}) \\
((\mathsf{x} : \mathsf{A}) \to \mathsf{B})^{\mathsf{P}} & = & \lambda\,\mathsf{f} \to ((\mathsf{x} : \mathsf{A})\,(\mathsf{x}^{\mathsf{M}} : \mathsf{A}^{\mathsf{P}}\,\mathsf{x}) \\
 & & \quad \to \mathsf{B}^{\mathsf{P}}\,(\mathsf{f\,x})) \\
(\lambda\,\mathsf{x} \to \mathsf{t})^{\mathsf{P}} & = & \lambda\,\mathsf{x}\,\mathsf{x}^{\mathsf{M}} \to \mathsf{t}^{\mathsf{P}} \\
(\mathsf{t\,u})^{\mathsf{P}} & = & \mathsf{t}^{\mathsf{P}}\,\mathsf{u}\,(\mathsf{u}^{\mathsf{P}}) \\
(\mathsf{t}\,[\,\sigma\,])^{\mathsf{P}} & = & \mathsf{t}^{\mathsf{P}}\,[\,\sigma^{\mathsf{P}}\,] \\
\epsilon^{\mathsf{P}} & = & \epsilon \\
(\sigma\,,\mathsf{x} \mapsto \mathsf{t})^{\mathsf{P}} & = & (\sigma^{\mathsf{P}}\,,\mathsf{x} \mapsto \mathsf{x},\,\mathsf{x}^{\mathsf{M}} \mapsto \mathsf{t}^{\mathsf{P}}) \\
(\sigma \circ \delta)^{\mathsf{P}} & = & \sigma^{\mathsf{P}} \circ \delta^{\mathsf{P}} \\
\mathsf{id}^{\mathsf{P}} & = & \mathsf{id}
\end{array}$$

5.2 Formal development

Using Agda, we formalise the above interpretation for the theory described in section 3. We express parametricity by a dependent function of the following type:

$$(\mathsf{t} : \mathsf{Tm}\,\Gamma\,\mathsf{A}) \to \mathsf{Tm}\,(\Gamma^{\mathsf{P}})\,(\mathsf{A}^{\mathsf{P}}\,[\,<\mathsf{t}\,[\,\mathsf{pr}\,\Gamma\,]\mathsf{t}>\,]\mathsf{T})$$

As t lives in context Γ, we need to substitute it by using a substitution $\mathsf{pr}\ \Gamma : \mathsf{Tms}\ (\Gamma^{\mathsf{P}})\ \Gamma$. A^{P} will not be a predicate function anymore but a type in an extended context, see below. Because the above given type is a dependent type, in contrast with the standard model, we cannot use the recursor to define this interpretation, we need the eliminator.

In constrast to the informal presentation of parametricity, here we have an additional judgement for types and we use de Bruijn indices instead of variable names which makes explicit weakening necessary. The motives for the eliminator are defined in figure 17. We need to specify the fields $\mathsf{Con}^{\mathsf{M}}$, Ty^{M}, $\mathsf{Tms}^{\mathsf{M}}$ and Tm^{M} in the record type Motives. We construct these components separately as $\mathsf{Con}^{\mathsf{m}}$, Ty^{m}, $\mathsf{Tms}^{\mathsf{m}}$ and Tm^{m}. $\mathsf{Con}^{\mathsf{m}}$ specifies what contexts will be mapped to: they will not only be mapped to the doubled context =C, but also to a weakening substitution Pr from the doubled context to the original one. We put these together into a record where =C and Pr are the projections, so if $\Gamma^{\mathsf{M}} : \mathsf{Con}^{\mathsf{M}}$ then $=\!\mathsf{C}\ \Gamma^{\mathsf{M}} : \mathsf{Con}$ and $\mathsf{Pr}\ \Gamma^{\mathsf{M}} : \mathsf{Tms}\ (=\!\mathsf{C}\ \Gamma^{\mathsf{M}})\ \Gamma$.

The motive for types receives the result Γ^{M} of the eliminator on the context and the type as A arguments. It will need to return a type in the context =C Γ^{M} extended with the type A (which needs to be substituted by the projection). The predicate over A is expressed by a type in a context extended with the domain of the predicate. A substitution ρ will be mapped to the interpretation =s which is between two doubled contexts and to a naturality property PrNat which expresses that =s commutes with Pr. This property is used eg. to define the method for _[_]T, see below. Terms will be mapped to terms in the doubled context and their type is the predicate at the original term (which has to be weakened by Pr): this is expressed by substituting the type by the term using <_> (defined in section 3). After defining the methods for this interpretation, the eliminator ElimTm will give us a proof of parametricity for this theory:

```
=C ∘ ElimCon  :   Con → Con
=s ∘ ElimTms  :   Tms Γ Δ
              →   Tms (=C (ElimCon Γ))
                      (=C (ElimCon Δ))
Pr ∘ ElimCon  :   (Γ : Con)
              →   Tms (=C (ElimCon Γ)) Γ
ElimTy        :   (A : Ty Γ)
              →   Ty  ( =C (ElimCon Γ)
                      , A [ Pr (ElimCon Γ) ]T)
ElimTm        :   (t : Tm Γ A)
              →   Tm (=s (ElimCon Γ))
                     ( ElimTy A
                       [ < t [ Pr (ElimCon Γ) ]t > ]T)
```

We list the methods for contexts and types as fields of the record Methods in figure 18. We omitted some implicit arguments and used record syntax more liberally than Agda. The other methods are straightforward but tedious to define due to the coercions that need to be performed. For details see the supplementary material.

The empty context is mapped to the empty context and the empty substitution. The context Γ , A is mapped to the doubled context =C Γ^{M} for Γ extended by A and the interpretation A^{M}. The projection substitution for Γ , A just projects out the A by lifting Pr Γ^{M} and is weakened so that it forgets about the additional A^{M} in the context.

For deriving the interpretation of a type A : Ty Δ substituted by δ : Tms Γ Δ we need to give a type in the context =C Γ^{M} , A [δ]T [Pr Γ^{M}]T by the motive for types. The type A^{M} lives in the context =C Δ^{M} , A [Pr Δ^{M}]T and by the interpretation of the substitution δ and lifting over A [Pr Δ^{M}]T by _ ↑ _ we get

```
=s δ^M ↑ A [ Pr Δ^M ]T
  : Tms (=C Γ^M , A [ Pr Δ^M ]T [ =s δ^M ]T)
        (=C Δ^M , A [ Pr Δ^M ]T) .
```

We can substitute A^{M} by =s δ^{M} ↑ A [Pr Δ^{M}]T but still we would get a type in the context

```
=C Γ^M , A [ Pr Δ^M ]T [ =s δ^M ]T
```

instead of

```
=C Γ^M , A [ δ ]T [ Pr Γ^M ]T.
```

However by the naturality rule PrNat δ^{M} we know that

```
Pr Δ^M ∘ =s δ^M ≡ δ ∘ Pr Γ^M .
```

With this in mind we can perform the following equality reasoning:

```
A [ Pr Δ^M ]T [ =s δ^M ]T
                          ≡⟨ [][]T                          ⟩
A [ Pr Δ^M ∘ =s δ^M ]T
                          ≡⟨ ap (_[_]T A) (PrNat δ^M) ⟩
A [ δ ∘ Pr Γ^M ]T
                          ≡⟨ [][]T ⁻¹                       ⟩
A [ δ ]T [ Pr Γ^M ]T
```

Coercing =s δ^{M} ↑ A [Pr Δ^{M}]T along this equality (we denote transitivity of equality by _ ▪ _) we get a substitution of the right type

```
Tms (=C Γ^M , A [ Pr Δ^M ]T [ =s δ^M ]T)
    (=C Δ^M , A [ Pr Δ^M ]T) .
```

Similar coercions are taking place everywhere in the interpretation. In the rest of the code-snippet we just write _ for the proofs of equalities for the coercions. The interpretation of U is like that of Set in the informal presentation: predicates over the type corresponding to the code in the last element of the context which can be projected by π_2 id. The predicate returns a code of a type in U. The interpretation of El goes the opposite way: it takes the predicate A^{M} from A into the universe and turns it into a predicate type by first using app to depend on the last element of the context and then applying El to get the type corresponding to the code. The interpretation of Π is the usual logical relation interpretation: we are in the context =C Γ^{M} , Π A B [Pr Γ^{M}]T and we would like to state that related arguments are mapped to related results by the function given in the last element of the context. We quantify over the argument type A (which needs to be weakened by Pr Γ^{M} because we are in an interpreted context, and by one step further because of the last element in the context) and then over A^{M} which depends on A so the weakening here over Π A B [Pr Γ^{M}]T needs to be lifted by _ ↑ _. The target of the function is B^{M} which lives in the context (=C (Γ^{M} ,C$^{\mathsf{M}}$ A^{M}) , B [Pr (Γ^{M} ,C$^{\mathsf{M}}$ A^{M})]T), the first part of which is provided by the substitution

```
π₁ id ↑ A [ Pr Γ^M ]T ↑ A^M
  : Tms ( =C Γ^M , Π A B [ Pr Γ^M ]T
        , A [ Pr Γ^M ]T [ π₁ id ]T
        , A^M [ π₁ id ↑ A [ Pr Γ^M ]T ]T)
        (=C Γ^M , A [ Pr Γ^M ]T , A^M)
```

which forgets the function from the context and is identity on the rest of the context and the second part is given by applying the function to the next element in the context and appropriately weakening and coercing the result.

Defining the logical predicate interpretation is tedious but feasible. Most of the work that needs to be done is coercing syntactic expressions using equality reasoning which can be simplified by using a heterogeneous equality [2] — two terms of different types are equal if there is an equality between the types and an equality between the terms up to this previous equality.

```
record _≃_ {A B : Set} (a : A) (b : B) : Set₁ where
  constructor _,_
  field
    projT : A ≡ B
    projt : a ≡[ projT ]≡ b
```

_≃_ can be proven to be reflexive, symmetric and transitive so equality reasoning can be done in the usual way. But it has the advantage that we can forget about coercions during reasoning:

```
uncoe : {a : A} (p : A ≡' B) → a ≃ coe p a
```

Also, we can convert back and forth with the usual homogeneous equality:

```
from≡ : (p : A ≡ B) → a ≡[ p ]≡ b → a ≃ b
```

```
m : Methods M
m = record
  {•^M = record {=C = •; Pr = ε}
  ; Γ^M ,C^M A^M
    = record { =C = (=C Γ^M , A [ Pr Γ^M ]T) , A^M
             ; Pr = (Pr Γ^M ↑ A) ∘ π₁ id}}
  ; A^M [ δ^M ]T^M
    = A^M [ coe (ap (λ σ → Tms (=C Γ^M , σ) _)
                    (  [][]T
                     · ap (_[_]T A) (PrNat δ^M)
                     · [][]T ⁻¹))
                  (=s δ^M ↑ A [ Pr Δ^M ]T) ]T
  ; U^M = Π (El (coe _ (π₂ id))) U
  ; El^M Â^M = El (app (coe _ Â^M))
  ; Π^M A^M B^M
    = Π (A [ Pr Γ^M ]T [ π₁ id ]T)
        (Π (A^M [ π₁ id ↑ A [ Pr Γ^M ]T ]T)
           (B^M [  π₁ id ↑ A [ Pr Γ^M ]T ↑ A^M
                ,  coe _ (app (coe _ (π₂ id))
                              [ π₁ id ]t) ]T))
  }
```

Figure 18. Methods for specifying the logical predicate interpretation for contexts and types

```
to≡   : (p : a ≃ b)   → a ≡[ projT p ]≡ b
```

5.3 The eliminator for closed inductive-inductive types

An application of the logical predicate interpretation is to derive the syntax of the motives and methods for the eliminator of a closed inductive-inductive type.

A general closed inductive-inductive type has the following description in Agda notation:

```
data A₁ : T₁  (signatures of types)
...
data Aₙ : Tₙ
data A₁ where (constructors for A₁)
  c₁₁    : A₁₁
  ...
  c₁ₘ₁   : A₁ₘ₁
...
data Aₙ where (constructors for Aₙ)
  cₙ₁    : Aₙ₁
  ...
  cₙₘₙ   : Aₙₘₙ
```

We have n types and the type A_i has m_i constructors. Agda restricts parameters of the constructors to only have strict positive recursive occurrences of the type. The same restriction applies here.

First we note that the above description can be collected into the context where the variable names are the type names and constructor names, and they have the corresponding types:

$$\bullet\,, A_1 : T_1\,, \ldots, A_n : T_n\,, c_{11} : A_{11}\,, \ldots, c_{1m_1} : A_{1m_1}\,, \ldots, c_{n1} : A_{n1}\,, \ldots, c_{nm_n} : A_{nm_n}$$

To define the motives and methods for the eliminator, we need a family over the types and fibers of that family over the constructors. By applying the $\_^{P}$ operation to this context, the context is extended by new elements ${A_1}^{M}, \ldots, {A_n}^{M}$ the types of which are the motives and by new elements ${c_{11}}^{M}, \ldots, {c_{nm_n}}^{M}$ the types of which will be the methods, and they can be listed in a record:

```
record Motives : Set where
  field
    A₁^M    : T₁ ^P A₁
    ...
    Aₙ^M    : Tₙ ^P Aₙ
record Methods (M : Motives) : Set where
  open Motives M
  field
    c₁₁^M   : A₁₁ ^P c₁₁
    ...
    cₙₘₙ^M  : Aₙₘₙ ^P cₙₘₙ
```

The method described here extends to types with equality constructors by using the logical predicate interpretation of the equality type. This is how we derived the motives and methods for the eliminator of the syntax.

6. Homotopy Type Theory

So far we have assumed uniqueness of identity proofs so let us have a look at what happens if we give this up to be compatible with Homotopy Type Theory as presented in [25]. If we take our definition of the Syntax and consider it as a HIT, we get a strange theory. Because we have not identified any of the equality constructors we introduced this leads to a very non-standard type theory. I.e. we may consider two types which have the same syntactic structure but which at some point use two different derivation to derive the same equality but these cannot be shown to be equal.

However, this can be easily remedied by *truncating* our syntax to be a set, i.e. by introducing additional constructors:

```
setT : {A B : Ty Γ}      {e0 e1 : A ≡ B} → e0 ≡ e1
sets : {δ σ : Tms Γ Δ}   {e0 e1 : δ ≡ σ} → e0 ≡ e1
sett : {u v : Tm Γ A}    {e0 e1 : u ≡ v} → e0 ≡ e1
```

These force our syntax to be a *set* in the sense of HoTT, i.e. a type for which UIP holds. We don't need to do this for Con because this can be shown to be a set from the assumption that Ty are sets. It seems to be entirely sensible to assume that the syntax forms a set, indeed we would want to show that equality is decidable which implies that the type is a set by Hedberg's theorem [13].

However, we now run into a different problem: we can only eliminate into a type which is a set itself. That means that we cannot even define the standard model because we have to eliminate into Set_1, the type of all small types, which is not a set in the sense of HoTT due to univalence, that is it has there may be more that one equality proof between two sets. One way around this would be to replace Set_1 by an inductive-recursive universe, which can be shown to be a set but for which univalence fails (see the formal development for the proofs).

```
data UU : Set
EL : UU → Set
data UU where
  'Π' : (A : UU) → (EL A → UU) → UU
  'Σ' : (A : UU) → (EL A → UU) → UU
  '⊤' : UU
EL ('Π' A B) = (x : EL A) → EL (B x)
EL ('Σ' A B) = Σ (EL A) λ x → EL (B x)
EL '⊤' = ⊤
```

An apparent way around the limitation that we can only eliminate into sets would be to only define the syntax in normal form and use a normalisation theorem. Since the normal forms do not require equality constructors there is no need to force the type to be a set and hence we could eliminate into any type. Indeed, this was proposed as a possible solution to the coherence problem in HoTT (e.g. how to define semi-simplicial types). However, it seems likely that this is not possible either. While we should be able to define the syntax of normal forms without equations we will need to incorporate normalisation. An example would be the rule for application for normal forms:

$$\begin{aligned} \_ \$ \_ : \quad & \mathsf{Ne}\,\Gamma\,(\Pi\,\mathsf{A}\,\mathsf{B}) \to (\mathsf{u} : \mathsf{Nf}\,\Gamma\,\mathsf{A}) \\ & \to \mathsf{Ne}\,\Gamma\,(\mathsf{B}\,[\,<\mathsf{u}>\,]\mathsf{T}) \end{aligned}$$

Here we assume that we mutually define normal Nf and neutral terms Ne and that all the types are in normal form. However, a problem is the substitution appearing in the result which has to substitute a normal term into a normal type giving rise to a normal type. This cannot be a constructor since then we would have to add equalities to specify how substitution has to be behave. Hence we have to execute the substitution and at the same time normalize the result (this is known as hereditary substitution [22]). We may still think that this may be challenging but possible using an inductive-recursive definition. However, even in the simplest case, i.e. in a type theory only with variables we have to prove equational properties of the explicit substitution operation, which in turn appear in the proof terms, leading to a coherence problem which we have so far failed to solve.

Nicolai Kraus raised the question whether it may be possible to give the interpretation of a strict model like the standard model (section 4) with the truncation even though we do not eliminate into a set. This is motivated by his work on general eliminations for the truncation operator [17]. Following this idea it may be possible to eliminate into set via an intermediate definition which states all the necessary coherence equations.

While defining the internal type theory as a set in HoTT seems to be of limited use, there are interesting applications in a 2-level theory similar to HTS as proposed by Voevodsky [26]. While the original proposal of HTS works in an extensional setting, it makes sense to consider a 2-level theory in an intensional setting like Agda. We start with a *strict* type theory with uniqueness of identity proofs (UIP) but within this we introduce a HoTT universe. This universe comes with its own propositional equality which is univalent but isn't proof-irrelevant. From this equality we can only eliminate into types within the universe. We call the types on the outside *pretypes* and the types in the universe *types*. The construction of the type-theoretic syntax takes place on the level of pretypes which is compatible with our assumption of UIP. On the other hand we can eliminate into the HoTT universe which is univalent. In this setting definitional equalities are modelled by strict equality and propositional equality by the univalent equality within the universe. Our definition of the syntax takes place at the level of pretypes but when constructing specific interpretations we eliminate into types.

7. Discussion and further work

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 [1] 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.

Clearly, we have only considered a very rudimentary type theory here, mainly for reasons of space. It is quite straightforward to add other type constructors, e.g. Σ-types, equality types, universes. We also 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 [16]. 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.

Acknowledgments

Thanks to Frederik Forsberg for discussions related to various aspects of the paper and joint work on the normal form problem of the pure version of internal type theory. We would also like to thank Paolo Capriotti, Gabe Dijkstra and Nicolai Kraus for discussions and work related to the topic of this paper, in particular questions related to HITs and coherence problems.

References

[1] T. Altenkirch, M. Hofmann, and T. Streicher. Categorical reconstruction of a reduction free normalization proof. In D. Pitt, D. E. Rydeheard, and P. Johnstone, editors, *Category Theory and Computer Science*, LNCS 953, pages 182–199, 1995.

[2] T. Altenkirch, C. McBride, and W. 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. ISBN 978-1-59593-677-6. .

[3] T. Altenkirch, P. Morris, F. N. Forsberg, and A. Setzer. A categorical semantics for inductive-inductive definitions. In *CALCO*, pages 70–84, 2011.

[4] J.-P. Bernardy, P. Jansson, and R. Paterson. Proofs for free — parametricity for dependent types. *Journal of Functional Programming*, 22(02):107–152, 2012. .

[5] M. Bezem, T. Coquand, and S. 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.

[6] E. Brady. Idris, a general-purpose dependently typed programming language: Design and implementation. *Journal of Functional Programming*, 23:552–593, 2013. ISSN 1469-7653. .

[7] M. Brown and J. Palsberg. Self-representation in Girard's System U. *SIGPLAN Not.*, 50(1):471–484, Jan. 2015. ISSN 0362-1340. . URL `http://doi.acm.org/10.1145/2775051.2676988`.

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

[9] J. Chapman. Type theory should eat itself. *Electron. Notes Theor. Comput. Sci.*, 228:21–36, Jan. 2009. ISSN 1571-0661. . URL `http://dx.doi.org/10.1016/j.entcs.2008.12.114`.

[10] N. A. Danielsson. A Formalisation of a Dependently Typed Language as an Inductive-Recursive Family. In *TYPES*, pages 93–109, 2006.

[11] D. Devriese and F. 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. ISBN 978-1-4503-2326-0. . URL `https://lirias.kuleuven.be/handle/123456789/404549`.

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

[13] M. Hedberg. A coherence theorem for Martin-Löf's type theory. *Journal of Functional Programming*, 8(04):413–436, 1998.

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

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

[16] G. Jaber, N. Tabareau, and M. Sozeau. Extending type theory with forcing. In *Logic in Computer Science (LICS), 2012 27th Annual IEEE Symposium on*, pages 395–404. IEEE, 2012.

[17] N. Kraus. *Truncation Levels in Homotopy Type Theory*. PhD thesis, University of Nottingham, 2015.

[18] D. Licata. Running circles around (in) your proof assistant; or, quotients that compute, 2011. available online.

[19] C. McBride. Outrageous but meaningful coincidences: dependent type-safe syntax and evaluation. In B. C. d. S. Oliveira and M. Zalewski, editors, *Proceedings of the ACM SIGPLAN Workshop on Generic Programming*, pages 1–12. ACM, 2010. ISBN 978-1-4503-0251-7. .

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

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

[22] F. Pfenning. Church and curry: Combining intrinsic and extrinsic typing. 2008.

[23] 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.

[24] The Agda development team. Agda, 2015. URL `http://wiki.portal.chalmers.se/agda`.

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

[26] V. Voevodsky. A type system with two kinds of identity types. Slides of a talk at the Institute for Advanced Study, February 2013.