A Rewrite System for Elimination of the Functional Extensionality and Univalence Axioms

Raphaël Cauderlier

Abstract

We explore the adaptation of the ideas that we developed for automatic elimination of classical axioms [5] to extensionality axioms in Type Theory.

Extensionality axioms are added to Type Theory when we lack a way to prove equality between two indistinguishable objects. These indistinguishable objects might be logically-equivalent propositions, pointwise equal functions, pointwise equivalent predicates, or equivalent (isomorphic) types leading to the various following extensionality axioms:

- Propositional extensionality: $PE : \Pi P, Q : \mathbf{Prop}.\ (P \Leftrightarrow Q) \to P =_{\mathbf{Prop}} Q$
- Non-dependent functional extensionality: $NFE : \Pi A, B : U.\ \Pi f, g : (A \to B).\ (\Pi a : A.\ f\ a =_B g\ a) \to f =_{A \to B} g$
- Dependent functional extensionality: $DFE : \Pi A : U.\ \Pi B : (A \to U).\ \Pi f, g : (\Pi a : A \to B(a)).\ (\Pi a : A.\ f\ a =_{B(a)} g\ a) \to f =_{\Pi a:A.\ B(a)} g$
- Predicate extensionality: $PredE : \Pi A : U.\ \Pi P, Q : (A \to \mathbf{Prop}).\ (\Pi a : A.\ P\ a \Leftrightarrow Q\ a) \to P =_{A \to \mathbf{Prop}} Q$
- Set extensionality: $SetE : \Pi A : U.\ \Pi X, Y : \mathsf{set}(A).\ (\Pi a : A.\ a \in X \Leftrightarrow a \in Y) \to X =_{\mathsf{set}(A)} Y$
- Univalence: let $\mathsf{idtoeqv}$ be the canonical function of type $\Pi A, B : U.\ (A =_U B) \to (A \approx B)$, for all types $A, B : U$, $\mathsf{idtoeqv}\ A\ B$ is an equivalence between the types $A =_U B$ and $A \approx B$.

In these axiom statements, the symbol $=_A$ denotes equality between two terms of type A, $\mathsf{set}(A)$ is type of the sets of elements of A, $\in$ is the set membership relation, U is a universe and $\simeq$ is the equivalence relation over types in U.

Set extensionality is essentially another notation for predicate extensionality. Obviously, $DFE \to NFE$ and $NFE \to PE \to PredE$. Less obviously, **Prop** can be defined in such a way that logical equivalence corresponds to type equivalence; with this definition, univalence implies PE. Even less obviously, univalence implies DFE (§§4.9 in [12]). In the context of elimination of the extensionality axioms, we probably do not want to reduce the relatively simple axiom of functional extensionality to the more complex axiom of univalence; we do however discard the axioms of propositional and predicate extensionality by considering **Prop** as a defined kind.

The univalence axiom has been proposed in the context of Homotopy Type Theory (HoTT) as a new foundation of mathematics. In HoTT (§§2.10 in [12]), the univalence axiom is stated as follows:

- If A and B are types, then $A =_U B \to A \simeq B$. (proof is easy)

- Axiom: The function idtoeqv constructed by this proof is itself an equivalence.

In particular, the inverse ua of idtoeqv has the following interesting type: $A \simeq B \to A =_U B$ which can be read as "undistinguishable types are in fact equal", which corresponds to a common habit in mathematics, in particular in category theory, of reasoning "modulo isomorphism".

Most mathematical concepts have multiple useful representations; for example:

- natural numbers may be represented in Peano form, which is useful for recursion, or in binary form which leads to more efficient extracted code,
- integers may be seen as pairs of a sign and an absolute value or as equivalence classes of differences of natural numbers,
- finite sets can be represented by sorted lists or red-black trees,
- polynomials can be represented as sets of roots or as arrays of coefficients,
- graphs can be represented as sets of arcs and vertices or as adjacent matrices,

It is often useful to play with the different representations of the same object to achieve some complex mathematical proof. Hence univalence is promising since once we have proven that two representations are equivalent, we can rewrite from one to the other thanks to the axiom. For example, if we prove a complex arithmetic result using Peano natural numbers (which have a nice inductive structure) and if this result has the form of a predicate independent of the representation applied to the type of natural numbers in Peano representation, then we obtain for free by univalence the corresponding theorem on the binary representation of natural numbers.

However, the univalence axiom is more than just the ua function, it is also the important fact that this function is the inverse of idtoeqv and so has a computational content. In particular:

- ua maps the identity function to the reflexivity of equality: $\mathsf{ua}(\mathsf{id}_A) = \mathsf{refl}_A$,
- ua maps composition of equivalences to transitivity of equality: $\mathsf{ua}(g \circ f) = \mathsf{trans}\ (\mathsf{ua}(f))\ (\mathsf{ua}(g))$
- ua maps equivalence inverses to symmetry of equality: $\mathsf{ua}(f^{-1}) = \mathsf{sym}\ (\mathsf{ua}(f))$.

Unfortunately, the axiomatic nature of the univalence makes these equalities only propositional. It would be interesting to work in a theory where *ua* is defined as a partial function that reduces by pattern matching on the function that it receives **and** on the types A and B.

In such a context, we expect the univalence function to be an algorithm that given a proof of an equivalence between two representations A and B and a proof of some property on A constructs the proof of the equivalent property on B.

In the rest of this section, we define in Dedukti a rewrite system representing a type theory featuring an extensional equality defined by ad-hoc polymorphism in Section 1. The main benefit of this rewrite system is that several extensionality axioms can uniformly be stated. This uniformity is exploited in Section 2 in which rewrite rules are added to eliminate the extensionality axioms.

1 Extensional Equality in Dedukti

In order to study the elimination of the extensionality axioms, we propose a Dedukti rewrite system in which equality is defined by ad-hoc polymorphism in such a way that functional extensionality and univalence hold by definition. In particular, we define the type $A =_U B$ as $A \simeq B$ when U is a universe. We present our rewrite system both in mathematical notations and in Dedukti syntax. The mathematical syntax is used to explain the meaning of the rewrite rules. We omit some type arguments, in particular for constructors of inductive types to improve readability, the missing pieces of information are given in the Dedukti terms. When type arguments are given, they are placed in subscript so for example we write $=_A$ instead of $=\ A$.

The ambient type theory in which we are developing this rewrite system is the fragment of Martin-Löf Type Theory in which we do not take the inductive definitions of natural numbers and equality. Natural numbers are not needed for our rewrite system and we want to give another definition of equality. Unfortunately, we have to take dependent products and dependent sums into consideration because the definition of $\simeq$ uses them.

In [12], several equivalent definitions of $\simeq$ are proposed, we choose the simplest. A function f is an equivalence if it has a left inverse (a function g such that $g \circ f$ is the identity) and a right inverse (a function g such that $g \circ f$ is the identity). Formally, the relation $\simeq$ is defined as follows:

$$\begin{aligned}(A \simeq B) := \quad & \Sigma f : A \to B. \\ & (\Sigma g : B \to A.\ \Pi x : A.\ g\ (f\ x) =_A x)\ \times (\Sigma g : A \to B.\ \Pi y : B.\ f\ (g\ y) =_B y)\end{aligned}$$

- We write U for any universe $\mathbf{Type}_i$, letting the universe level implicit following the common habit of *typical ambiguity*. In Dedukti, types are represented as usual:

  ```
  #NAME ua.

  (; Types ;)
  type : Type.
  def term : type -> Type.
  ```

  The universe level is also omitted on the Dedukti side

  ```
  (; Universe ;)
  U : type.
  [] term U --> type.
  ```

  Girard's paradox [6] can be expressed in this encoding if the implicit universe levels are used in an inconsistent manner so a serious application of this work would require to make the universe levels explicit but it is orthogonal to the presentation of our rewrite system.

  When using mathematical notations, we do not distinguish between a term of type `type` and the type of its inhabitants : we omit the function `term` completely. Nor do we distinguish between `type` and `U`, a universe is simply written U.

- The empty type 0 has no constructor:

  ```
  (; Empty type ;)
  0 : Type.
  Zero : type.
  [] term Zero --> 0.
  ```

- The singleton type 1 has a constructor $0_1 : 1$:

```
(; Singleton type ;)
1 : Type.
One : type.
[] term One --> 1.
0_1 : 1.
```

- Cartesian product is written $A \times B$, its constructor is written (a, b):

```
(; Product ;)
prod : type -> type -> Type.
Prod : type -> type -> type.
[A,B] term (Prod A B) --> prod A B.
Pair : A : type -> B : type -> term A -> term B -> prod A B.
```

- Sum-type is written $A + B$, its constructors are $\mathsf{left} : A \to A + B$ and $\mathsf{right} : B \to A + B$:

```
(; Sum ;)
sum : type -> type -> Type.
Sum : type -> type -> type.
[A,B] term (Sum A B) --> sum A B.
Left : A : type -> B : type -> term A -> sum A B.
Right : A : type -> B : type -> term B -> sum A B.
```

- Arrow type is written $A \to B$, its constructor is λ:

```
(; Dependend Product ;)
Arr : type -> type -> type.
[A, B] term (Arr A B) --> term A -> term B.
```

- Dependent product is written $\Pi x : A.\ B$ where x is bound in B, the constructor of dependent product is also written λ:

```
(; Dependend Product ;)
Pi : A : type -> (term A -> type) -> type.
[A, B] term (Pi A B) --> x : term A -> term (B x).
```

- Dependent sum is written $\Sigma x : A.\ B$ where x is bound in B, the constructor of dependent sum is also written (a, b):

```
(; Dependend Sum ;)
sig : A : type -> (term A -> type) -> Type.
Sig : A : type -> (term A -> type) -> type.
[A, B] term (Sig A B) --> sig A B.
Dpair : A : type -> B : (term A -> type) ->
        a : term A -> term (B a) -> sig A B.
```

We proceed to our definition of equality case by case, defining $=_A$ on a type A amounts to provide a type for $a =_A a'$ when a and a' are obtained from the constructors of type A:

```
def Eq : A : type -> term A -> term A -> type.
def eq : A : type -> term A -> term A -> Type.
[A,x,y] eq A x y --> term (Eq A x y).
```

- 0 has no constructor so there is nothing to do to define $=_0$.
- 1 has exactly one constructor 0_1 and it should be trivially equal to itself:

$$(0_1 =_1 0_1) := 1$$

  ```
  [] Eq One 0_1 0_1 --> One
  ```

- two inhabitants of $A + B$ are equal if they are equal terms of type A or equal terms of type B, they are different otherwise:

$$\begin{aligned}&(\mathsf{left}\ a =_{A+B} \mathsf{left}\ a') := (a =_A a')\\&(\mathsf{left}\ a =_{A+B} \mathsf{right}\ b) := 0\\&(\mathsf{right}\ b =_{A+B} \mathsf{left}\ a) := 0\\&(\mathsf{right}\ b =_{A+B} \mathsf{right}\ b') := (b =_B b')\end{aligned}$$

  ```
  [A,B,a,a'] Eq (Sum A B) (Left _ _ a) (Left _ _ a') --> Eq A a a'
  [A,B,a,b] Eq (Sum A B) (Left _ _ a) (Right _ _ b) --> Zero
  [A,B,b,a] Eq (Sum A B) (Right _ _ b) (Left _ _ a) --> Zero
  [A,B,b,b'] Eq (Sum A B) (Right _ _ b) (Right _ _ b') --> Eq B b b'
  ```

- two inhabitants of $A \times B$ are equal if they are componentwise equal:

$$((a, b) =_{A\times B} (a', b')) := (a =_A a') \times (b =_B b')$$

  ```
  [A,B,a,b,a',b'] Eq (Prod A B) (Pair _ _ a b) (Pair _ _ a' b') -->
     Prod (Eq A a a') (Eq B b b')
  ```

- two inhabitants of $A \to B$ are equal if they are pointwise equal hence our definition of equality satisfies non-dependent functional extensionality by definition:

$$(f =_{A\to B} g) := \Pi x : A.\ f\ x =_B g\ x$$

  ```
  [A,B,f,g] Eq (Arr A B) f g --> Pi A (x => Eq B (f x) (g x))
  ```

- similarly, two inhabitants of $\Pi x : A.\ B(x)$ are equal is they are pointwise equal hence our definition of equality satisfies dependent functional extensionality by definition:

$$(f =_{\Pi x:A.\ B(x)} g) := \Pi x : A.\ f\ x =_{B(x)} g\ x$$

```
[A,B,f,g] Eq (Pi A B) f g --> Pi A (x => Eq (B x) (f x) (g x)).
```

- two types are equal if they are equivalent hence our definition of equality satisfies univalence by definition:

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

```
def Are_semi_inverse : A : type -> B : type ->
                       term (Arr A B) -> term (Arr B A) -> type.
[A,B,f,g] Are_semi_inverse A B f g --> Pi A (x => Eq A (g (f x)) x).

def Isequiv : A : type -> B : type -> term (Arr A B) -> type.
[A,B,f] Isequiv A B f -->
   Prod (Sig (Arr B A) (g => Are_semi_inverse A B f g))
        (Sig (Arr B A) (g => Are_semi_inverse B A g f)).

def Eqv : type -> type -> type.
[A,B] Eqv A B --> Sig (Arr A B) (Isequiv A B).
def eqv : type -> type -> Type.
[A,B] eqv A B --> term (Eqv A B).

[A,B] Eq U A B --> Eqv A B.
```

- the only remaining case is dependent sum; given two dependent pairs (a, b) and (a', b') of type $\Sigma x : A.\ B(x)$, we start by comparing a and a' in type A. Since b and b' do not have convertible types, we use our equality between a and a' to substitute a by a' in the type of b; this operation is called transport:

$$((a, b) =_{\Sigma x:A.\ B(x)} (a', b')) := \Sigma p : a =_A a'.\ (\mathsf{transport}_A\ a\ a'\ p\ B\ b) =_{B(a')} b'$$

The type of $\mathsf{transport}$ is

$$\mathsf{transport} : \Pi A : U.\ \Pi a : A.\ \Pi a' : A.\ (a =_A a') \to\ \Pi P : (A \to U).\ P\ a \to P\ a'$$

```
def transport : A : type -> a : term A -> a' : term A -> eq A a a' ->
                P : (term A -> type) -> term (P a) -> term (P a').
[A,B,a,b,a',b'] Eq (Sig A B) (Dpair _ _ a b) (Dpair _ _ a' b') -->
      Sig (Eq A a a') (p => Eq (B a') (transport A a a' p B b) b').
```

The transport operation has to be defined mutually with equality, defining $\mathsf{transport}$ on a type A amounts to provide an inhabitant of $\Pi P : (A \to U).\ P\ a \to P\ a'$ when a and a' are obtained from the constructors of type A and p is obtained from the constructors of type $a =_A a'$:

- there is no inhabitant in 0 so there is nothing to do for $\mathsf{transport}_0$.
- there is a single constructor 0_1 in 1,

$$\mathsf{transport}_1\ 0_1\ 0_1\ 0_1 := \lambda P : (1 \to U).\ \lambda h : P\ 0_1.\ h$$

```
[h] transport One 0_1 0_1 0_1 _ h --> h
```

- to define $\mathsf{transport}_{A+B}$, we only have to give two cases because there are no proofs of equality in the two other cases:

$$\begin{aligned}&\mathsf{transport}_{A+B}\ (\mathsf{left}\ a)\ (\mathsf{left}\ a')\ p :=\\ &\quad \lambda P : (A+B \to U).\ \lambda h : P(\mathsf{left}\ a).\ \mathsf{transport}_A\ a\ a'\ p\ (\lambda x : A.\ P(\mathsf{left}\ x))\ h\\ &\mathsf{transport}_{A+B}\ (\mathsf{right}\ b)\ (\mathsf{right}\ b')\ p :=\\ &\quad \lambda P : (A+B \to U).\ \lambda h : P(\mathsf{right}\ b).\ \mathsf{transport}_B\ b\ b'\ p\ (\lambda x : B.\ P(\mathsf{right}\ x))\ h\end{aligned}$$

```
[A,B,a,a',p,P,h]
   transport (Sum A B) (Left _ _ a) (Left _ _ a') p P h
     -->
   transport A a a' p (x => P (Left A B x)) h
[A,B,b,b',p,P,h]
   transport (Sum A B) (Right _ _ b) (Right _ _ b') p P h
     -->
   transport B b b' p (x => P (Right A B x)) h
```

- to define $\mathsf{transport}_{A\times B}$, we use the transport operations on A and B in sequence:

$$\begin{aligned}&\mathsf{transport}_{A\times B}\ (a,b)\ (a',b')\ (p_A,p_B) :=\\ &\quad \lambda P : (A\times B \to U).\ \lambda h : P(a,b).\\ &\qquad \mathsf{transport}_B\ b\ b'\ p_B\ (\lambda y : B.\ P(a',y))\ (\mathsf{transport}_A\ a\ a'\ p_A\ (\lambda x : A.\ P(x,b))\ h)\end{aligned}$$

```
[A,B,a,b,a',b',P,pA,pB,h]
   transport (Prod A B) (Pair _ _ a b) (Pair _ _ a' b')
             (Pair _ _ pA pB) P h
     -->
   transport B b b' pB (y => P (Pair A B a' y))
      (transport A a a' pA (x => P (Pair A B x b)) h).
```

- transport is not definable on arrows, dependent products and universes because these cases correspond to unprovable axioms.

- the only remaining case is again dependent sum and is again the most complicated; to substitute a dependent pair (a,b) by an equal one (a',b') in a term $h : P(a,b)$, we first substitute a by a' leading to a term $\tilde{h} : P(a', \mathsf{transport}_A\ a\ a'\ B\ p\ b)$ but this requires to extend transport to allow the predicate to depend on the inhabitant of the equality type.

This generalized transport is called based-path induction (this name comes from the homotopy interpretation of equality as path, see [12]):

$$\begin{aligned}\mathsf{bpi} :\quad & \Pi A : U.\ \Pi a : A.\ \Pi a' : A.\ \Pi p : (a =_A a').\ \Pi P : (\Pi x : A.\ (a =_A x) \to U).\\ & \quad P\ a\ (\mathsf{refl}_A\ a) \to P\ a'\ p\end{aligned}$$

where refl states the reflexivity of equality:

$$\mathsf{refl} : \Pi A : U.\ \Pi a : A.\ (a =_A a)$$

We define both bpi and refl by ad-hoc polymorphism again:

```
def Refl : A : type -> a : term A -> eq A a a.

def bpi : A : type -> a : term A -> a' : term A -> p : eq A a a' ->
          P : (x : term A -> eq A a x -> type) ->
          term (P a (Refl A a)) -> term (P a' p).
```

- there is nothing to do for 0.
- the definitions are straightforward for 1:

$$\begin{aligned}&\mathsf{refl}_1\ 0_1 := 0_1\\ &\mathsf{bpi}_1\ 0_1\ 0_1\ 0_1 := \lambda P : (\Pi x : 1.\ (0_1 =_1 x) \to U).\ \lambda h : P\ 0_1\ 0_1.\ h\end{aligned}$$

```
[] Refl One 0_1 --> 0_1.
[h] bpi One 0_1 0_1 0_1 _ h --> h.
```

- for type $A + B$, we only have two cases to consider:

$$\begin{aligned}&\mathsf{refl}_{A+B}\ (\mathsf{left}\ a) := \mathsf{refl}_A\ a\\ &\mathsf{refl}_{A+B}\ (\mathsf{right}\ b) := \mathsf{refl}_B\ b\\ &\mathsf{bpi}_{A+B}\ (\mathsf{left}\ a)\ (\mathsf{left}\ a')\ p_A :=\\ &\qquad \lambda P : (\Pi x : A.\ (a =_A x) \to U).\\ &\qquad \lambda h : P\ (\mathsf{left}\ a)\ (\mathsf{refl}_A\ a).\\ &\qquad\qquad \mathsf{bpi}_A\ a\ a'\ p_A\ (\lambda x : A.\ \lambda p : a =_A x.\ P\ (\mathsf{left}\ x)\ p)\ h\\ &\mathsf{bpi}_{A+B}\ (\mathsf{right}\ b)\ (\mathsf{right}\ b')\ p_B :=\\ &\qquad \lambda P : (\Pi y : B.\ (b =_A y) \to U).\\ &\qquad \lambda h : P\ (\mathsf{right}\ b)\ (\mathsf{refl}_B\ b).\\ &\qquad\qquad \mathsf{bpi}_B\ b\ b'\ p_B\ (\lambda y : B.\ \lambda p : b =_B y.\ P\ (\mathsf{right}\ y)\ p)\ h\end{aligned}$$

```
[A,B,a] Refl (Sum A B) (Left _ _ a) --> Refl A a
[A,B,b] Refl (Sum A B) (Right _ _ b) --> Refl B b.
[A,B,a,a',pA,P,h] bpi (Sum A B) (Left _ _ a) (Left _ _ a') pA P h -->
    bpi A a a' pA (x => p => P (Left A B x) p) h
[A,B,b,b',pB,P,h] bpi (Sum A B) (Right _ _ b) (Right _ _ b') pB P h -->
    bpi B b b' pB (x => p => P (Right A B x) p) h.
```

- again, bpi can not be defined for arrows, dependent products and universes but we can easily define reflexivity in these cases:

$$\begin{aligned}&\mathsf{refl}_{A \to B}\ f := \lambda x : A.\ \mathsf{refl}_B\ (f\ x)\\ &\mathsf{refl}_{\Pi x:A.\ B(x)}\ f := \lambda x : A.\ \mathsf{refl}_{B(x)}\ (f\ x)\\ &\mathsf{refl}_U\ A := (\lambda x : A.\ x, ((\lambda x : A.\ x, \mathsf{refl}_A), (\lambda x : A.\ x, \mathsf{refl}_A)))\end{aligned}$$

```
[A,B,f] Refl (Arr A B) f --> a : term A => Refl B (f a)
[A,B,f] Refl (Pi A B) f --> a : term A => Refl (B a) (f a)
[A] Refl U A -->
    Dpair (Arr A A) (Isequiv A A) (x => x)
      (Pair (Sig (Arr A A) (g => Are_semi_inverse A A (x => x) g))
            (Sig (Arr A A) (g => Are_semi_inverse A A g (x => x)))
            (Dpair (Arr A A) (g => Are_semi_inverse A A (x => x) g)
               (x => x) (Refl A))
            (Dpair (Arr A A) (g => Are_semi_inverse A A g (x => x))
               (x => x) (Refl A))).
```

- last but not least, we define refl and bpi for products and Σ-types; $\mathsf{refl}_{\Sigma x:A.\ B(x)}\ (a,b)$ should have type $\Sigma p : a =_A a.\ (\mathsf{transport}_A\ a\ a\ p\ B\ b) =_{B(a)} b$, the only reasonable choice to inhabit $a =_A a$ is to take $\mathsf{refl}_A\ a$ and we are then asked to inhabit the type $(\mathsf{transport}_A\ a\ a\ (\mathsf{refl}_A\ a)\ B\ b) =_{B(a)} b$. This is not a problem in usual type theory where $\mathsf{transport}_A\ a\ a\ (\mathsf{refl}_A\ a)\ B\ b$ is convertible to b; to allow a similar conversion, we add the rewrite rule

$$\mathsf{transport}_A\ a\ a\ (\mathsf{refl}_A\ a)\ P\ h \longrightarrow h$$

```
[A,a,P,h] transport A a a (Refl A a) P h --> h.
```

This rewrite rule is not linear, we would like to replace it by its linearization:

$$\mathsf{transport}_A\ a\ a''\ (\mathsf{refl}_{A'}\ a')\ P\ h \longrightarrow h$$

letting Dedukti infer that $a \equiv a' \equiv a''$ and $A \equiv A'$ in order to type-check the rewrite rule. Unfortunately, this depends on the injectivity of $=$ which Dedukti has no mean to guarantee. In fact, nothing prevents the user to add non-injective rewrite rules between this one and the moment where it will be matched so Dedukti is right in rejecting the linear rule.

For this reason, the current development has been type-checked by Dedukti using the non-linearity flag and could not be checked for confluence automatically.

We define refl on products and Σ-types by

$$\begin{aligned}\mathsf{refl}_{A\times B}\ (a,b) &:= (\mathsf{refl}_A\ a, \mathsf{refl}_B\ b)\\ \mathsf{refl}_{\Sigma x:A.\ B(x)}\ (a,b) &:= (\mathsf{refl}_A\ a, \mathsf{refl}_{B(a)}\ b)\end{aligned}$$

```
[A,B,a,b] Refl (Prod A B) (Pair _ _ a b) -->
      Pair (Eq A a a) (Eq B b b) (Refl A a) (Refl B b)
[A,B,a,b] Refl (Sig A B) (Dpair _ _ a b) -->
      Dpair (Eq A a a)
            (p : eq A a a => Eq (B a) (transport A a a p B b) b)
            (Refl A a) (Refl (B a) b).
```

In order to define bpi, we generalize the non-linear rule on transport to the dependent case:

$$\mathsf{bpi}_A\ a\ a\ (\mathsf{refl}_A\ a)\ P\ h \longrightarrow h$$

```
[A,a,P,h] bpi A a a (Refl A a) P h --> h.
```

This allows us to define bpi. In order to shorten this definition and increase readability, we use local definitions. These local definitions are unfolded in the actual Dedukti file.

$$
\begin{aligned}
&\mathsf{bpi}_{A\times B}\ (a,b)\ (a',b')\ (p_A,p_B) := \\
&\quad \lambda P : (\Pi c : A\times B.\ ((a,b) =_{A\times B} c) \to U). \\
&\quad \lambda h : P\ (a,b)\ (\mathsf{refl}_A\ a, \mathsf{refl}_B\ b). \\
&\qquad \mathsf{bpi}_A\ a\ a'\ p_A\ (\lambda x : A.\ \lambda p_x : a =_A x.\ P\ (x,b')\ (p_x,p_B)) \\
&\qquad\quad (\mathsf{bpi}_B\ b\ b'\ p_B\ (\lambda y : B.\ \lambda p_y : b =_B y.\ P\ (a,y)\ (\mathsf{refl}_A\ a, p_y))\ h)
\end{aligned}
$$

$$
\begin{aligned}
&\mathsf{bpi}_{\Sigma x:A.\ B(x)}\ (a,b)\ (a',b')\ (p_A,p_B) := \\
&\quad \lambda P : (\Pi c : \Sigma x : A.\ B(x).((a,b) = c) \to U). \\
&\quad \lambda h : P\ (a,b)\ (\mathsf{refl}_A\ a, \mathsf{refl}_{B(a)}\ b). \\
&\qquad \textbf{let}\ \mathit{move}\ (x : A)\ (p_x : a =_A x) : B(x) := \mathsf{transport}_A\ a\ x\ p_x\ B\ b\ \textbf{in} \\
&\qquad \textbf{let}\ \tilde{b} : B(a') := \mathit{move}\ a'\ p_A\ \textbf{in} \\
&\qquad \textbf{let}\ \tilde{h} : P\ (a',\tilde{b})\ (p_A, \mathsf{refl}_{B(a')}\ \tilde{b}) := \\
&\qquad\quad \mathsf{bpi}_A\ a\ a'\ p_A\ (\lambda x : A.\ \lambda p_x : a =_A x.P\ (x, \mathit{move}\ x\ p_x)\ (p_x, \mathsf{refl}_{B(x)}\ (\mathit{move}\ x\ p_x)))\ h \\
&\qquad \textbf{in} \\
&\qquad \mathsf{bpi}_{B(a')}\ \tilde{b}\ b'\ p_B\ (\lambda y : B(a').\ \lambda p_y : \tilde{b} =_{B(a')} y.\ P\ (a',y)\ (p_A,p_y))\ \tilde{h}
\end{aligned}
$$

```
[A,B,a,b,a',b',P,pA,pB,h]
   bpi (Prod A B) (Pair _ _ a b) (Pair _ _ a' b') (Pair _ _ pA pB) P h
     -->
   bpi B b b' pB
     (b'' => p =>
      P (Pair A B a' b'') (Pair (Eq A a a') (Eq B b b'') pA p))
     (bpi A a a' pA
        (a' => pA =>
         P (Pair A B a' b) (Pair (Eq A a a') (Eq B b b) pA (Refl B b)))
        h).

[A,B,a,b,a',b',P,pA,pB,h]
   bpi (Sig A B) (Dpair _ _ a b) (Dpair _ _ a' b') (Dpair _ _ pA pB) P h
     -->
   bpi (B a') (transport A a a' pA B b) b' pB
     (b'' => p =>
      P (Dpair A B a' b'')
        (Dpair (Eq A a a')
               (pA => Eq (B a') (transport A a a' pA B b) b'') pA p))
     (bpi A a a' pA
        (a' => pA =>
         P (Dpair A B a' (transport A a a' pA B b))
           (Dpair (Eq A a a')
              (p =>
                      Eq (B a') (transport A a a' p B b) (transport A a a' pA B b))
                     pA
                     (Refl (B a') (transport A a a' pA B b))))
           h).
```

Let us summarize what we have done; we have defined extensional equality by ad-hoc polymorphism (by providing a different definition for each possible type constructor) recursively and mutually with the reflexivity theorem and the dependent elimination scheme bpi. The only missing parts in this definition correspond to the three extensionality axioms: non-dependent functional extensionality, dependent functional extensionality, and univalence. In other words, we have stated these three axioms in a uniform way:

- $\mathsf{transport}_{A\to B} : \Pi f : A \to B.\ \Pi g : A \to B.\ (\Pi x : A.\ f\ x =_B g\ x) \to \Pi P : ((A \to B) \to U).\ P\ f \to P\ g$
- $\mathsf{transport}_{\Pi x:A.\ B(x)} : \Pi f : (\Pi x : A.\ B(x)).\ \Pi g : (\Pi x : A.\ B(x)).\ (\Pi x : A.\ f\ x =_{B(x)} g\ x) \to \Pi P : ((\Pi x : A.\ B(x)) \to U).\ P\ f \to P\ g$
- $\mathsf{transport}_U : \Pi A : U.\ \Pi B : U.\ (A \simeq B) \to \Pi P : (U \to U).\ P\ A \to P\ B$

2 Extending the Computational Content

We can now look for additional rewrite rules to eliminate these axioms. The first rules that we should add are those which close the critical pairs with the rewrite rule
$\mathsf{transport}_A\ a\ a\ (\mathsf{refl}_A\ a)\ P\ h \longrightarrow h$:

- $\mathsf{transport}_{A\to B}\ f\ f\ (\lambda x : A.\ \mathsf{refl}_B\ (f\ x))\ P\ h \longrightarrow h$
- $\mathsf{transport}_{\Pi x:A.\ B(x)}\ f\ f\ (\lambda x : A.\ \mathsf{refl}_{B(x)}\ (f\ x))\ P\ h \longrightarrow h$
- $\mathsf{transport}_U\ A\ A\ (\lambda x : A.\ x, ((\lambda x : A.\ x, \mathsf{refl}_A), (\lambda x : A.\ x, \mathsf{refl}_A)))\ P\ h \longrightarrow h$

```
[B,f,h] bpi (Arr _ B) f f (x => Refl B (f x)) _ h --> h
[B,f,h] bpi (Pi _ B) f f (x => Refl (B x) (f x)) _ h --> h.
[A,h] bpi U A A
   (Dpair (Arr A A) (Isequiv A A) (x => x)
      (Pair _ _
        (Dpair (Arr A A) _ (x => x) (Refl A))
        (Dpair (Arr A A) _ (x => x) (Refl A)))) _ h --> h.

[B,f,h] transport (Arr _ B) f f (x => Refl B (f x)) _ h --> h
[B,f,h] transport (Pi _ B) f f (x => Refl (B x) (f x)) _ h --> h.
[A,h] transport U A A
   (Dpair (Arr A A) (Isequiv A A) (x => x)
      (Pair _ _
        (Dpair (Arr A A) _ (x => x) (Refl A))
        (Dpair (Arr A A) _ (x => x) (Refl A)))) _ h --> h.
```

Even if it is very tempting, we do not add the general rewrite rule $\mathsf{transport}_A\ a\ a\ \_\ P\ h \longrightarrow h$ because is would lead to inconsistency since in Homotopy Type Theory there can be equality proofs distinct from reflexivity. For example, there are two proofs of $2 =_U 2$ where 2 is the type $1 + 1$, the first one is $\mathsf{refl}_U\ 2$ and the second one is $e := (\mathsf{switch}, ((\mathsf{switch}, \mathsf{switch\text{-}auto}), (\mathsf{switch}, \mathsf{switch\text{-}auto})))$ where $\mathsf{switch} : 2 \to 2$ is defined by $\mathsf{switch}\ (\mathsf{left}\ 0_1) := \mathsf{right}\ 0_1$ and $\mathsf{switch}\ (\mathsf{right}\ 0_1) := \mathsf{left}\ 0_1$ and $\mathsf{switch\text{-}auto} : \Pi x : 2.\ \mathsf{switch}\ (\mathsf{switch}\ x) =_2 x$ is defined by $\mathsf{switch\text{-}auto}\ (\mathsf{left}\ 0_1) := \mathsf{refl}_2\ (\mathsf{left}\ 0_1)$ and $\mathsf{switch\text{-}auto}\ (\mathsf{right}\ 0_1) := \mathsf{refl}_2\ (\mathsf{right}\ 0_1)$. $\mathsf{transport}_U\ 2\ 2\ e$ should not be the identity but the operation consisting of exchanging left and right.

We can play on which argument we match against in the rewrite rules:

$$
\begin{array}{l}
\mathsf{transport}_U\ A\ B\ (e, \_)\ (\lambda X : U.\ X)\ a \longrightarrow e\ a \\
\mathsf{transport}_A\ a\ a'\ \_\ (\lambda x : A.\ B)\ b \longrightarrow b \\
\mathsf{transport}_A\ a\ a'\ p\ (\lambda x : A.\ B(x) \times C(x))\ (b, c) \longrightarrow (\mathsf{transport}_A\ a\ a'\ p\ B\ b, \mathsf{transport}_A\ a\ a'\ p\ C\ c) \\
\mathsf{transport}_A\ a\ a'\ p\ (\lambda x : A.\ B(x) + C(x))\ \mathsf{left} b \longrightarrow \mathsf{left}(\mathsf{transport}_A\ a\ a'\ p\ B\ b) \\
\mathsf{transport}_A\ a\ a'\ p\ (\lambda x : A.B(x) + C(x))\ \mathsf{right} c \longrightarrow \mathsf{right}(\mathsf{transport}_A\ a\ a'\ p\ C\ c) \\
\mathsf{transport}_A\ a\ a'\ p\ (\lambda x : A.\ B(x) \to C(x))\ f \longrightarrow \\
\qquad \lambda b : B(a').\ \mathsf{transport}_A\ a\ a'\ p\ C\ (f(\mathsf{transport}_A\ a'\ a\ p^{-1}\ B\ b))
\end{array}
$$

In the last rule, p^{-1} is the inverse of p resulting from the application of the symmetry of equality. This lemma can be defined either from transport or directly by induction over types if we want to limit the dependency to transport.

```
def Sym : A : term U -> a : term A -> b : term A ->
          term (Eq A a b) -> term (Eq A b a).
[] Sym One 0_1 0_1 0_1 --> 0_1
[A,B,a,a',p] Sym (Sum A B) (Left _ _ a) (Left _ _ a') p --> Sym A a a' p
[A,B,b,b',p] Sym (Sum A B) (Right _ _ b) (Right _ _ b') p --> Sym B b b' p
[A,B,f,g,p] Sym (Arr A B) f g H --> a => Sym B (f a) (g a) (p a)
[A,B,f,g,p] Sym (Pi A B) f g H --> a => Sym (B a) (f a) (g a) (p a)
[A,B,a,b,a',b',pA,pB]
   Sym (Prod A B) (Pair _ _ a b) (Pair _ _ a' b') (Pair _ _ pA pB)
     -->
   Pair (Eq A a' a) (Eq B b' b) (Sym A a a' pA) (Sym B b b' pB).

[A,B,e,a]
   transport U A B (Dpair (Arr A B) (Isequiv A B) e _) (X => X) a --> e a
[B,b] transport _ _ _ _ (x => B) b --> b
[A,a,a',B,C,p,b,c]
   transport A a a' p (x => Prod (B x) (C x)) (Pair _ _ b c)
     -->
   Pair (B a') (C a')
     (transport A a a' p (x => B x) b) (transport A a a' p (x => C x) c)
[A,a,a',B,C,D,p,f] transport A a a' p (x => Arr (B x) (C x)) f -->
   b => transport A a a' p (x => C x)
       (f (transport A a' a (Sym A a a' p) (x => B x) b)).
```

Similar rules can be added for dependent product, dependent sum, and equality but they are a bit complicated so we did not implement them. For the same reason, we did not implement similar rules for the dependent version bpi either.

3 Related Work

For the particular case of representation change between the unary and the binary representations of natural numbers, Magaud [9] proposes a method for automatically retrieving theorems on the binary representation of natural numbers from an arithmetic library. He implemented this approach on a Coq unary arithmetic library and successfully produced the corresponding binary arithmetic library. The main difficulty in this work relies on the implicit use of computation through Coq conversion rule because the computational behaviour is not the same on the unary and the binary definition of arithmetic operations. The solution proposed by Magaud consists in automatically expliciting these computation steps. For highly computational proofs such as the Coq proof of the four colour theorem [7], this might however not be tractable. Contrary to Magaud's approach, we respect the Poincaré principle [3] of separation of reasoning and computing and we let the computational part of the proofs implicit. Magaud's work would be a good starting point for evaluating the efficiency of our rewrite system as a tool for automatic representation change.

Defining functions and reasoning by induction over types in Type Theory has a long story. In their presentation of Martin-Löf Type Theory [11], Nordstrom, Petersson, and Smith consider an elimination rule for the universe which closes the universe but allows for reasoning by induction on the structure of types. Development of proof assistants for Type Theory such as Lego, Coq, and Agda then droped this principle in favor of open universes in which inductive types can be added at will. The NuPRL proof assistant for extensional MLTT [10] also follows this approach but its Oyster2 derivative [8] came back to type induction in order to define tactics in Type Theory itself instead of an auxiliary meta-language (usually a variant of prolog or ML). As far as we know, Oyster2 is the only implementation of close universes and type induction.

Most of the second chapter of the HoTT book [12] is dedicated to characterizing equality on type constructors. Functional extensionality and the univalence axiom are motivated by the lack of such characterization for dependent product and universes. However, the book complains about the lack of computational behaviour of these characterizations; in HoTT, these characterizations of equalities are not judgmental but only propositional.

In order to prove the consistency of the axiom of functional extensionality in type theory, an extensional model of type theory can be defined using setoids []. This model is internalized in Observational Type Theory [1]. In OTT, a few constructs are added to intentional type theory (called coercions, heterogeneous equality, and coherence operator) in order to get a provably extensional equality while preserving canonicity (the property that closed normal forms begin with a constructor). However, both the setoid model and OTT require a strong notion of proof irrelevance: if φ is a proposition, all the terms of type φ should be convertible. It is still unclear whether or not proof irrelevant type theory can be encoded Dedukti (this issue is discussed in [2] in the context of the encoding of Matita to Dedukti) so it is hard to relate our work with OTT.

Similarly for the univalence axiom, a model of univalent type theory called the cubical model has been internalized in type theory leading to Cubical Type Theory [4], an extension of HoTT in which univalence is derivable from other operations (glue and unglue) in a way which gives it more computational behaviour. This work is a lot more ambitious than our approach. In particular, Cubical Type Theory is believed to have the canonicity property.

References

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

[2] Ali Assaf. *A Framework for Defining Computational Higher-Order Logics.* PhD thesis, École Polytechnique, 2015.

[3] Henk Barendregt and Erik Barendsen. Autarkic Computations in Formal Proofs. *Journal of Automated Reasoning (JAR)*, 28, 2002.

[4] Lars Birkedal, Ales Bizjak, Ranald Clouston, Hans Bugge Grathwohl, Bas Spitters, and Andrea Vezzosi. Guarded Cubical Type Theory: Path Equality for Guarded Recursion. *CoRR*, abs/1606.05223, 2016.

[5] Raphaël Cauderlier. A rewrite system for proof constructivization. In *Proceedings of the 2016 International Workshop on Logical Frameworks and Meta-languages: Theory and Practice.* ACM, 2016. To appear.

[6] Jean-Yves Girard. *Interprétation fonctionnelle et élimination des coupures de l'arithmétique d'ordre supérieur.* PhD thesis, Université Paris 7, 1972.

[7] Georges Gonthier. The Four Colour Theorem: Engineering of a Formal Proof. In Deepak Kapur, editor, *Computer Mathematics, 8th Asian Symposium, ASCM 2007, Singapore, December 15-17, 2007. Revised and Invited Papers*, volume 5081 of *Lecture Notes in Computer Science*, page 333. Springer, 2007.

[8] Christian Horn and Alan Smaill. From Meta-level Tactics to Object-level Programs. In Jeffrey Johnson, Sean McKee, and Alfred Vella, editors, *Artificial Intelligence in Mathematics*, pages 135–146. Oxford University Press, Inc., New York, NY, USA, 1994.

[9] Nicolas Magaud and Yves Bertot. Changing data structures in type theory: A study of natural numbers. In Paul Callaghan, Zhaohui Luo, James McKinna, and Robert Pollack, editors, *Types for Proofs and Programs, International Workshop, TYPES 2000, Durham, UK, December 8-12, 2000, Selected Papers*, volume 2277 of *Lecture Notes in Computer Science*, pages 181–196. Springer, 2000.

[10] Per Martin-Löf. Constructive Mathematics and Computer Programming. In L. Jonathan Cohen, Jerzy Łoś, Helmut Pfeiffer, and Klaus-Peter Podewski, editors, *Proceedings of the Sixth International Congress of Logic, Methodology and Philosophy of Science*, volume 104 of *Studies in Logic and the Foundations of Mathematics.* North-Holland, 1982.

[11] Bengt Nordstrom, Kent Petersson, and Jan M Smith. Programming in Martin-Lof's Type Theory. An Introduction. In *Number 7 in International series of monographs on computer science.* Oxford University Press, 1989.

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