Isomorphism Is Equality

Thierry Coquand   Nils Anders Danielsson

Abstract

The setting of this work is dependent type theory extended with the univalence axiom. We prove that, for a large class of algebraic structures, isomorphic instances of a structure are equal—in fact, isomorphism is in bijective correspondence with equality. The class of structures includes monoids whose underlying types are "sets", and also posets where the underlying types are sets and the ordering relations are pointwise "propositional". For instance, equality of monoids on sets coincides with the usual notion of isomorphism from universal algebra, and equality of posets of the kind mentioned above coincides with order isomorphism.

1 Introduction

De Bruijn argued that it is more natural for mathematicians to work with a typed language than with the untyped universe of set theory (1975). In this paper we explore a possible *mathematical* advantage of working in a type theory—inspired by the ones designed by de Bruijn and his coworkers[1] (de Bruijn 1980)—over working in set theory.

Consider the following two monoids:

$$(\mathbb{N}, \lambda mn.\ m + n, 0)$$

and

$$(\mathbb{N} \setminus \{ 0 \}, \lambda mn.\ m + n - 1, 1).$$

These monoids are *isomorphic*, as witnessed by the isomorphism $\lambda n.\ n + 1$. However, in set theory they are not *equal*: there are properties that are satisfied by only one of them. For instance, only the first one satisfies the property that the carrier set contains the element 0.

In (a certain) type theory extended with the univalence axiom (see Section 2) the situation is different. This is the focus of the present paper:

- We prove that monoids M_1 and M_2 that are isomorphic, i.e. for which there is a homomorphic bijection $f : M_1 \to M_2$, are equal (see Section 3.5). In fact, we show that isomorphism is in bijective correspondence with equality.

  Note that the equality that we use is substitutive. This means that, unlike in set theory, any property that holds for the first monoid above also holds for the second one.

[1]The AUTOMATH project team included van Benthem Jutting, van Daalen, Kornaat, Nederpelt, de Vrijer, Zandleven, Zucker, and others (Nederpelt and Geuvers 1994).

(The result is restricted to monoids whose carrier types are “sets”. This term is defined in Section 2.5. Many types, including the natural numbers, are sets.)

- The result about monoids is an instance of a more general theorem (see Section 3.3), which applies to a large class of algebraic structures, including posets and discrete fields (defined as in Section 3.5).

All the main results in the paper have been formalised using the proof assistant Agda[2] (Norell 2007; Agda Team 2013), which is based on Martin-Löf type theory (Martin-Löf 1975; Nordström et al. 1990). Unlike in regular Martin-Löf type theory we use a “non-computing” J rule (i.e. the computation rule for J only holds propositionally, not definitionally); this choice, which makes the result more generally applicable, is motivated in Section 2.3. We believe that our arguments carry over to other variants of type theory, but do not make any formal claims in this direction.

Note that our theorem is proved *inside* the type theory, using the univalence axiom. In the absence of this axiom we can still observe, *meta-theoretically*, that we cannot prove any statement that distinguishes the two monoids above (given the consistency of the axiom). A related observation was made already in the 1930s by Lindenbaum and Tarski (1983, see also Tarski (1986)): in one variant of type theory every sentential function is invariant under bijections.

The formulation of “isomorphism is equality” that is used in this paper is not intended to be as general as possible; we try to strike a good balance between generality and ease of understanding. Other variations of this result have been developed concurrently by Aczel and Shulman (2013). See Section 4 for further discussion of related work.

2 Preliminaries

This section introduces some concepts, terminology and results used below. We assume some familiarity with type theory.

The presentation below is close to the Agda formalisation, but differs in minor details. In particular, we do not always use proper Agda syntax.

2.1 Hierarchy of Types

We assume that we have an infinite hierarchy of “types of types” $\mathit{Type}_0 : \mathit{Type}_1 : \mathit{Type}_2 : \ldots$ (and use the synonym $\mathit{Type} = \mathit{Type}_0$). Below we define some concepts using certain types Type_i and Type_j. These definitions are applicable to arbitrary “universe levels” i and j.

In Agda a member of Type_i is not automatically a member of Type_j for $i < j$, but one can manually lift types from one level to another. In this paper we omit such liftings.

[2]Using the `--without-K` flag; the code of the formalisation can currently be downloaded from `http://www.cse.chalmers.se/~nad/`.

2.2 Quantifiers

If we have $A : \mathit{Type}_i$ and $B : A \to \mathit{Type}_j$, then we can introduce the Π-type, or dependent function type, $(x : A) \to B\ x$ (sometimes written $\forall\ x.\ B\ x$). If we have $f : (x : A) \to B\ x$ and $t : A$, then the application $f\ t$ has type $B\ t$. Simple (non-dependent) function types are written $A \to B$.

In order to reduce clutter we sometimes use "implicit" function types. The notations $\{x : A\} \to B\ x$ and $\forall\ \{x\}\ .\ B\ x$ mean the same as $(x : A) \to B\ x$ and $\forall\ x.\ B\ x$, respectively, except that the function's argument is not given explicitly: we write f rather than $f\ y$, with the intention that y can be inferred from the context.

Sometimes we combine several quantifiers into one: $(x\ y : A) \to B\ x\ y$ means the same as $(x : A) \to (y : A) \to B\ x\ y$, and $\forall\ x\ \{y\ z\}\ .\ B\ x\ y\ z$ means the same as $\forall\ x.\ \forall\ \{y\}\ .\ \forall\ \{z\}\ .\ B\ x\ y\ z$.

Σ-types, or dependent pairs, are written $\Sigma\ x : A.\ B\ x$ (or $\Sigma\ x.\ B\ x$). If we have $t : A$ and $u : B\ t$, then (t, u) has type $\Sigma\ x : A.\ B\ x$. Σ-types come with two projection functions. The first projection is written proj_1 and the second proj_2. Cartesian products (non-dependent pairs) are defined as $A \times B = \Sigma\ \_ : A.\ B$.

We make use of η-equality for both Π-types and Σ-types: the function $f : (x : A) \to B\ x$ is *definitionally* equal to $\lambda\ x.\ f\ x$ (where x is not free in f), and the pair $p : \Sigma\ x : A.\ B\ x$ is definitionally equal to $(\mathit{proj}_1\ p, \mathit{proj}_2\ p)$. (Definitional equality is discussed below.) We suspect that the use of η-equality is not essential, but have used it in our formalisation.

2.3 Equality

Following de Bruijn (1975) we distinguish between definitional (or judgemental) and propositional (or book) equality. Definitional equality ($\beta\eta$-equality plus unfolding of user-made definitions) is inferred automatically by the type checker, and comes with no term formers. If we have $t : A$, and A is definitionally equal to B, then we have $t : B$ as well: definitional equalities are "invoked automatically". Propositional equality, on the other hand, is a type with corresponding term formers.

The propositional equality type, containing proofs of equality between x and y, is written $x \equiv y$. Here x and y must both have the same type A, with $A : \mathit{Type}_i$. There is one introduction rule for equalities:

$$\mathsf{refl} : \{A : \mathit{Type}_i\} \to (x : A) \to x \equiv x$$

The corresponding eliminator is traditionally called J:

$$\begin{aligned} J : {} & \{A : \mathit{Type}_i\} \to \\ & (P : (x\ y : A) \to x \equiv y \to \mathit{Type}_j) \to \\ & (\forall\ x.\ P\ x\ x\ (\mathsf{refl}\ x)) \to \\ & \forall\ \{x\ y\}\ .\ (\mathit{eq} : x \equiv y) \to P\ x\ y\ \mathit{eq} \end{aligned}$$

Typically J and refl come together with a "computation rule", a definitional equality stating how applications of the form $J\ P\ r\ (\mathsf{refl}\ x)$ compute (Martin-Löf 1975). We include such a rule, but stated as a *propositional* equality:

$$\begin{aligned} J\text{-}\mathsf{refl}\ :\ & \{A : \mathit{Type}_i\} \to \\ & (P : (x\ y : A) \to x \equiv y \to \mathit{Type}_j) \to \\ & (r : \forall\, x.\ P\ x\ x\ (\mathsf{refl}\ x)) \to \\ & \forall\, x.\ J\ P\ r\ (\mathsf{refl}\ x) \equiv r\ x \end{aligned}$$

The reason for using a propositional computation rule is the ongoing quest to find a computational interpretation of the univalence axiom: perhaps we will end up with a computational interpretation in which J-refl does not hold definitionally.

As mentioned in the introduction the propositional equality type is substitutive. This follows directly from the J rule:

$$\begin{aligned} & \mathit{subst}\ :\ \{A : \mathit{Type}_i\} \to (P : A \to \mathit{Type}_j) \to \\ & \qquad\quad \{x\ y : A\} \to x \equiv y \to P\ x \to P\ y \\ & \mathit{subst}\ P = J\ (\lambda\ u\ v\ \_.\ P\ u \to P\ v)\ (\lambda\ \_\ p.\ p) \end{aligned}$$

We sometimes make use of axioms stating that propositional equality of functions is extensional:

$$\begin{aligned} \mathit{Extensionality}\ :\ & (A : \mathit{Type}_i) \to (B : A \to \mathit{Type}_j) \to \\ & (f\ g : (x : A) \to B\ x) \to \\ & (\forall\, x.\ f\ x \equiv g\ x) \to f \equiv g \end{aligned}$$

When we use the term "extensionality" below we refer to extensionality of functions. In the formalisation we explicitly pass around assumptions of extensionality ($\mathit{foo} : \mathit{Extensionality} \to \ldots$), thus making it clear when this assumption is *not* used. To avoid clutter we do not do so below.

The type of bijections between the types $A : \mathit{Type}_i$ and $B : \mathit{Type}_j$ is written $A \leftrightarrow B$. This type can be defined as a nested Σ-type:

$$\begin{aligned} A \leftrightarrow B = {} & \Sigma\ \mathit{to} : A \to B.\ \Sigma\ \mathit{from} : B \to A. \\ & (\forall\, x.\ \mathit{to}\ (\mathit{from}\ x) \equiv x) \times (\forall\, x.\ \mathit{from}\ (\mathit{to}\ x) \equiv x) \end{aligned}$$

If we have $f : A \leftrightarrow B$, then we use the notation $\mathit{to}\ f$ for the "forward" function of type $A \to B$, and $\mathit{from}\ f$ for the "backward" function of type $B \to A$.

A key property of equality of Σ-types is that equality of pairs p, q of type $\Sigma\ x : A.\ B\ x$ is in bijective correspondence with pairs of equalities:

$$p \equiv q \quad \leftrightarrow \quad \Sigma\ \mathit{eq} : \mathit{proj}_1\ p \equiv \mathit{proj}_1\ q.\ \mathit{subst}\ B\ \mathit{eq}\ (\mathit{proj}_2\ p) \equiv \mathit{proj}_2\ q$$

This property can be proved using J and J-refl. By assuming extensionality we can prove a similar key property of equality of Π-types (Voevodsky 2011):

$$f \equiv g \quad \leftrightarrow \quad \forall\, x.\ f\ x \equiv g\ x$$

2.4 More Types

The unit type is denoted $\top$ (with sole element $\mathsf{tt} : \top$), and the empty type $\bot$. Agda comes with η-equality for $\top$: all values of this type are definitionally equal.

The binary (disjoint) sum of the types A and B is written $A + B$. If we have $t : A$ and $u : B$, then we also have $\mathsf{inj}_1\ t : A + B$ and $\mathsf{inj}_2\ u : A + B$.

The natural numbers are defined as an inductive data type $\mathbb{N}$ with constructors $\mathsf{zero} : \mathbb{N}$ and $\mathsf{suc} : \mathbb{N} \to \mathbb{N}$. Natural numbers can be eliminated using structural recursion.

Two types A and B are *logically equivalent*, written $A \Leftrightarrow B$, if there are functions going from A to B and back: $A \Leftrightarrow B = (A \to B) \times (B \to A)$.

2.5 Univalent Foundations

Let us now introduce some terminology and results from the "univalent foundations of mathematics", largely but not entirely based on work done by Voevodsky (2010, 2011), and verified to apply in our setting (with a propositional computation rule for J).

Contractibility is defined as follows:

$$\begin{aligned}
&\mathit{Contractible} : \mathit{Type}_i \to \mathit{Type}_i \\
&\mathit{Contractible}\ A = \Sigma\, x : A.\ \forall\, y.\ x \equiv y
\end{aligned}$$

Homotopy levels or *h-levels* are defined by recursion on a natural number:

$$\begin{aligned}
&\mathit{H\text{-}level} : \mathbb{N} \to \mathit{Type}_i \to \mathit{Type}_i \\
&\mathit{H\text{-}level}\ \mathsf{zero}\quad A = \mathit{Contractible}\ A \\
&\mathit{H\text{-}level}\ (\mathsf{suc}\ n)\ A = (x\ y : A) \to \mathit{H\text{-}level}\ n\ (x \equiv y)
\end{aligned}$$

Types at level 0 are contractible. We call types at level 1 *propositions* and types at level 2 *sets*:

$$\begin{aligned}
&\mathit{Is\text{-}proposition} : \mathit{Type}_i \to \mathit{Type}_i \\
&\mathit{Is\text{-}proposition} = \mathit{H\text{-}level}\ 1 \\
&\mathit{Is\text{-}set} : \mathit{Type}_i \to \mathit{Type}_i \\
&\mathit{Is\text{-}set} = \mathit{H\text{-}level}\ 2
\end{aligned}$$

The following results can be used to establish that a type has a certain h-level:

- A type which has h-level n also has h-level $\mathsf{suc}\ n$.
- A type A is contractible iff it is in bijective correspondence with the unit type: $\top \leftrightarrow A$.
- A type A is a proposition iff all its values are equal: $(x\ y : A) \to x \equiv y$. In particular, $\bot$ is a proposition.
- A type A is a set iff it satisfies the "uniqueness of identity proofs" property (UIP): $(x\ y : A) \to (p\ q : x \equiv y) \to p \equiv q$.
- If a type A has decidable equality, $(x\ y : A) \to (x \equiv y) + (x \equiv y \to \bot)$, then it satisfies UIP (Hedberg 1998), so it is a set. In particular, $\mathbb{N}$ is a set.
- $\mathit{H\text{-}level}\ n\ A$ is a proposition (assuming extensionality).

- If A has h-level n, and, for all x, $B\ x$ has h-level n, then $\Sigma\ x : A.\ B\ x$ has h-level n.
- If, for all x, $B\ x$ has h-level n, then $(x : A) \to B\ x$ has h-level n (assuming extensionality).
- If A and B both have h-level n, where $n \geqslant 2$, then $A + B$ has h-level n.
- If A has h-level $n \geqslant 1$, then $W\ x : A.\ B\ x$ has h-level n (assuming extensionality). Here $W\ x : A.\ B\ x$ is a W-type, or well-founded tree type (Nordström et al. 1990).
- When proving that a type A has a positive h-level one can assume that A is inhabited: $(A \to \textit{H-level}\,(\mathsf{suc}\ n)\ A) \to \textit{H-level}\,(\mathsf{suc}\ n)\ A$.
- If there is a "split surjection" from A to B (i.e. a triple consisting of two functions $\textit{to} : A \to B$ and $\textit{from} : B \to A$ along with a proof of $\forall\ x.\ \textit{to}\ (\textit{from}\ x) \equiv x$), and A has h-level n, then B has h-level n.

If a type is known to be propositional, then one can use this knowledge to simplify equalities involving this type; propositional second components of pairs can be dropped:

$$
\begin{array}{l}
(p\ q : \Sigma\ x : A.\ B\ x) \to \\
\textit{Is-proposition}\,(B\ (\textit{proj}_1\ q)) \to \\
(p \equiv q) \leftrightarrow (\textit{proj}_1\ p \equiv \textit{proj}_1\ q)
\end{array}
$$

A function is an *equivalence* if all its "preimages" are contractible:

$$
\begin{array}{l}
\textit{Is-equivalence} : \{A\ B : \textit{Type}_i\} \to (A \to B) \to \textit{Type}_i \\
\textit{Is-equivalence}\,f = \forall\ y.\ \textit{Contractible}\ (\Sigma\ x.\ f\ x \equiv y)
\end{array}
$$

Observe that $\textit{Is-equivalence}\,f$ is a proposition (assuming extensionality). One example of an equivalence is $\textit{subst}\ P\ \textit{eq} : P\ x \to P\ y$ (for any P, x, y and $\textit{eq} : x \equiv y$).

Two types A and B are *equivalent*, written $A \simeq B$, if there is an equivalence from A to B:

$$\Sigma\ f : A \to B.\ \textit{Is-equivalence}\,f$$

If we have $\textit{eq} : A \simeq B$, then we use the (overloaded) notation $\textit{to eq}$ for the first projection of $\textit{eq}$. Given $\textit{eq} : A \simeq B$ it is also easy to construct a function of type $B \to A$. We use the overloaded notation $\textit{from eq}$ for this function: $\textit{from eq} = \lambda\ y.\ \textit{proj}_1\ (\textit{proj}_1\ (\textit{proj}_2\ \textit{eq}\ y))$.

One can prove that $\textit{to eq}$ and $\textit{from eq}$ are inverses, so $A \simeq B$ is logically equivalent to $A \leftrightarrow B$. When A is a set we can, assuming extensionality, strengthen this logical equivalence to a bijection: $(A \simeq B) \leftrightarrow (A \leftrightarrow B)$. If both A and B are propositions, then we can take this one step further—in this case equivalences are, again assuming extensionality, in bijective correspondence with logical equivalences: $(A \simeq B) \leftrightarrow (A \Leftrightarrow B)$.

The following "congruence" property illustrates one way in which one can prove that two types $\Sigma\ x : A.\ B\ x$ and $\Sigma\ x : C.\ D\ x$ are equivalent:

$$\begin{aligned}&(\mathit{eq} : A \simeq C) \to (\forall\, x.\, B\, x \simeq D\, (\mathit{to}\ \mathit{eq}\ x)) \to \\ &(\Sigma\, x : A.\, B\, x) \simeq (\Sigma\, x : C.\, D\, x)\end{aligned}$$

If we assume extensionality, then we can prove a corresponding property for Π-types:

$$\begin{aligned}&(\mathit{eq} : A \simeq C) \to (\forall\, x.\, B\, x \simeq D\, (\mathit{to}\ \mathit{eq}\ x)) \to \\ &((x : A) \to B\, x) \simeq ((x : C) \to D\, x)\end{aligned}$$

Similar properties can be proved for other type formers as well.

It is easy to show that equality implies equivalence:

$$\begin{aligned}&{\equiv}{\Rightarrow}{\simeq} : (A\ B : \mathit{Type}_i) \to A \equiv B \to A \simeq B \\ &{\equiv}{\Rightarrow}{\simeq}\ \_\ \_ = J\, (\lambda\, A\, B\, \_.\, A \simeq B)\, (\lambda\, \_ \to \mathit{id})\end{aligned}$$

(Here *id* is the identity equivalence.) The *univalence axiom* states that this function is an equivalence:

$$\mathit{Univalence} : (A\ B : \mathit{Type}_i) \to \mathit{Is\text{-}equivalence}\, ({\equiv}{\Rightarrow}{\simeq}\ A\ B)$$

As immediate consequences of the univalence axiom we get that equality is in bijective correspondence with equivalence, $(A \equiv B) \leftrightarrow (A \simeq B)$, and that we can convert equivalences to equalities:

$${\simeq}{\Rightarrow}{\equiv} : \{A\ B : \mathit{Type}_i\} \to A \simeq B \to A \equiv B$$

The univalence axiom (two instances, one at level j and one at level $j + 1$) also implies extensionality (at levels i and j). Furthermore univalence (at level i) can be used to prove the *transport theorem*:

$$\begin{aligned}&(P : \mathit{Type}_i \to \mathit{Type}_j) \to \\ &(\mathit{resp} : \{A\ B : \mathit{Type}_i\} \to A \simeq B \to P\ A \to P\ B) \to \\ &(\mathit{resp\text{-}id} : \forall\, A.\, (p : P\ A) \to \mathit{resp}\ \mathit{id}\ p \equiv p) \to \\ &\forall\, A\ B.\, (\mathit{eq} : A \simeq B) \to (p : P\ A) \to \\ &\mathit{resp}\ \mathit{eq}\ p \equiv \mathit{subst}\ P\ ({\simeq}{\Rightarrow}{\equiv}\ \mathit{eq})\ p\end{aligned}$$

This theorem states that if we have a function *resp* that witnesses that a predicate P respects equivalence, and *resp id* is the identity function, then *resp eq* is pointwise equal to $\mathit{subst}\ P\ ({\simeq}{\Rightarrow}{\equiv}\ \mathit{eq})$. By using the fact that $\mathit{subst}\ P\ ({\simeq}{\Rightarrow}{\equiv}\ \mathit{eq})$ is an equivalence we get that *resp eq* is also an equivalence, and that it preserves compositions (if we, in addition to univalence, assume extensionality).

We mentioned above that we make use of a global assumption of extensionality in the text. We also make use of a global assumption of univalence. To be precise, below we use univalence at the first three universe levels. These three instances of univalence can be used to prove all instances of extensionality that we make use of.

3 Isomorphism Is Equality

In this section we prove that isomorphism is equality for a large class of algebraic structures. First we prove the result for arbitrary "universes" satisfying certain properties, then we define a universe that is closed under function spaces, cartesian products, and binary sums, and finally we give some examples.

3.1 Parameters

We parametrise the general result by four components. The first two form a universe, i.e. a type U of codes, along with a decoding function *El*:

$$\begin{aligned}&U \ : \mathit{Type}_2\\ &\mathit{El} \ : U \ \rightarrow \ \mathit{Type}_1 \ \rightarrow \ \mathit{Type}_1\end{aligned}$$

We have chosen to use Type_2 and Type_1 (rather than, say, *Type* and *Type*) in order to support the example universe given in Section 3.4. However, other choices are possible.

The third component is a requirement that *El a*, when seen as a predicate, respects equivalences:

$$\mathit{resp} \ : \forall \ a \ \{B \ C\} \ . \ B \ \simeq \ C \ \rightarrow \ \mathit{El} \ a \ B \ \rightarrow \ \mathit{El} \ a \ C$$

Finally the *resp* function should map the identity equivalence *id* to the identity function:

$$\mathit{resp\text{-}id} \ : \forall \ a \ B. \ (x \ : \mathit{El} \ a \ B) \ \rightarrow \ \mathit{resp} \ a \ \mathit{id} \ x \ \equiv \ x$$

The idea is that an element $a \ : \ U$ corresponds to a kind of structure, that *El a B* is the type of elements having this structure and using the "carrier type" B, and that the operation *resp* corresponds to "transport of structure" (Bourbaki 1957): if $x \ : \mathit{El} \ a \ B$ and $\mathit{eq} \ : B \ \simeq \ C$ then $\mathit{resp} \ a \ \mathit{eq} \ x$ is the a-structure on C obtained by transporting x along *eq*.

3.2 Codes for Structures

Given these parameters we define a notion of codes for "extended" structures. The codes consist of two parts, a code in U and a family of propositions:

$$\begin{aligned}&\mathit{Code} \ : \mathit{Type}_3\\ &\mathit{Code} \ =\\ &\quad \Sigma \ a \ : \ U.\\ &\quad (C \ : \ \mathit{Type}_1) \ \rightarrow \ \mathit{El} \ a \ C \ \rightarrow \ \Sigma \ P \ : \ \mathit{Type}_1. \ \mathit{Is\text{-}proposition} \ P\end{aligned}$$

The codes are decoded in the following way (values of type *Instance c* are instances of the structure coded by c):

```
Instance : Code → Type₂
Instance (a, P) =
  Σ C : Type₁.    -- Carrier type.
  Σ x : El a C.   -- Element.
  proj₁ (P C x)   -- The element satisfies the corresponding
                  -- proposition.
```

We can also define what it means for two instances to be isomorphic. First we use *resp* to define a predicate that specifies when a given equivalence is an isomorphism from one element to another:

$$\begin{aligned}&\mathit{Is\text{-}isomorphism} \ : \forall \ a \ \{B \ C\} \ . \ B \ \simeq \ C \ \rightarrow \ \mathit{El} \ a \ B \ \rightarrow \ \mathit{El} \ a \ C \ \rightarrow \ \mathit{Type}_1\\ &\mathit{Is\text{-}isomorphism} \ a \ \mathit{eq} \ x \ y \ = \ \mathit{resp} \ a \ \mathit{eq} \ x \ \equiv \ y\end{aligned}$$

Two instances are then defined to be isomorphic if there is an equivalence between the carrier types that relates the elements; the propositions are ignored:

$$\begin{array}{l} \mathit{Isomorphic} : \forall\ c.\ \mathit{Instance}\ c \to \mathit{Instance}\ c \to \mathit{Type}_1 \\ \mathit{Isomorphic}\ (a, \_)\ (C, x, \_)\ (D, y, \_) = \\ \quad \Sigma\ \mathit{eq} : C \simeq D.\ \mathit{Is\text{-}isomorphism}\ a\ \mathit{eq}\ x\ y \end{array}$$

The following projections, one for carrier types and one for elements, are easy to define:

$$\begin{array}{l} \mathit{Carrier} : \forall\ c.\ \mathit{Instance}\ c \to \mathit{Type}_1 \\ \mathit{element} : \forall\ c.\ (I : \mathit{Instance}\ c) \to \mathit{El}\ (\mathit{proj}_1\ c)\ (\mathit{Carrier}\ c\ I) \end{array}$$

We use the projections to state that equality of instances is in bijective correspondence with a pair of equalities, one for the carrier types and one for the elements:

$$\begin{array}{l} \mathit{equality\text{-}pair\text{-}lemma} : \\ \quad \forall\ c.\ (I\ J : \mathit{Instance}\ c) \to \\ \quad (I \equiv J) \\ \qquad \leftrightarrow \\ \quad \Sigma\ \mathit{eq} : \mathit{Carrier}\ c\ I \equiv \mathit{Carrier}\ c\ J. \\ \qquad \mathit{subst}\ (\mathit{El}\ (\mathit{proj}_1\ c))\ \mathit{eq}\ (\mathit{element}\ c\ I) \equiv \mathit{element}\ c\ J \end{array}$$

Our proof of this statement is straightforward. Assume that $c = (a, P)$, $I = (C, x, p)$ and $J = (D, y, q)$. We proceed by "bijectional reasoning" (note that $\leftrightarrow$ is a transitive relation):

$$\begin{array}{lll} (C, x, p) & \equiv (D, y, q) & \leftrightarrow \\ ((C, x), p) & \equiv ((D, y), q) & \leftrightarrow \\ (C, x) & \equiv (D, y) & \leftrightarrow \\ \multicolumn{3}{l}{\Sigma\ \mathit{eq} : C \equiv D.\ \mathit{subst}\ (\mathit{El}\ a)\ \mathit{eq}\ x \equiv y} \end{array}$$

In the first step we apply a bijection to both sides of the equality, in the second step we drop the propositional second components of the tuples, and the last step uses the key property of equality of Σ-types that was mentioned in Section 2.3.

3.3 Main Theorem

Let us now prove the main result:

$$\mathit{isomorphism\text{-}is\text{-}equality} : \forall\ c\ I\ J.\ \mathit{Isomorphic}\ c\ I\ J \leftrightarrow (I \equiv J)$$

Assume that $c = (a, P)$, $I = (C, x, p)$ and $J = (D, y, q)$. As above we proceed by bijectional reasoning (after unfolding some definitions):

$$\begin{array}{lll} \Sigma\ \mathit{eq} : C \simeq D.\ \mathit{resp}\ a\ \mathit{eq}\ x & \equiv y & \leftrightarrow \\ \Sigma\ \mathit{eq} : C \simeq D.\ \mathit{subst}\ (\mathit{El}\ a)\ ({\simeq}{\Rightarrow}{\equiv}\ \mathit{eq})\ x & \equiv y & \leftrightarrow \\ \Sigma\ \mathit{eq} : C \equiv D.\ \mathit{subst}\ (\mathit{El}\ a)\ \mathit{eq}\ x & \equiv y & \leftrightarrow \\ I \equiv J & & \end{array}$$

The first step uses the transport theorem instantiated with *resp* and *resp-id*, the second step univalence, and the last step *equality-pair-lemma*.

An immediate corollary of *isomorphism-is-equality* (and univalence) is that *Isomorphic c I J* is equal to $I \equiv J$: isomorphism is equality.

3.4 A Universe

Let us now define a concrete universe. The codes and the decoding function are defined as follows:

```
  data U  : Type₂ where
    id    : U                    -- The argument.
    type  : U                    -- Type.
    k     : Type₁ → U            -- A constant.
    _⇾_   : U → U → U            -- Function space.
    _⊗_   : U → U → U            -- Cartesian product.
    _⊕_   : U → U → U            -- Binary sum.
  El : U → Type₁ → Type₁
  El id      C = C
  El type    C = Type
  El (k A)   C = A
  El (a ⇾ b) C = El a C → El b C
  El (a ⊗ b) C = El a C × El b C
  El (a ⊕ b) C = El a C + El b C
```

Here U is an inductive data type, with constructors id, type, k, etc., and *El a* is defined by recursion on the structure of a. The notation _⇾_ is used to declare an infix operator: the underscores mark the argument positions.

We do not define *resp* directly, instead we define a "cast" operator that shows that *El a* preserves logical equivalences:

```
  cast : ∀ a {B C} . B ⇔ C → El a B ⇔ El a C
```

The cast operator is defined by recursion on the structure of the code a:

```
  cast id      eq = eq
  cast type    eq = id
  cast (k A)   eq = id
  cast (a ⇾ b) eq = cast a eq →-eq cast b eq
  cast (a ⊗ b) eq = cast a eq ×-eq cast b eq
  cast (a ⊕ b) eq = cast a eq +-eq cast b eq
```

Here *id* is the identity logical equivalence. We omit the definitions of the logical equivalence combinators; they have the following types (for arbitrary types A, B, C, D):

```
  _→-eq_ : A ⇔ B → C ⇔ D → (A → C) ⇔ (B → D)
  _×-eq_ : A ⇔ B → C ⇔ D → (A × C) ⇔ (B × D)
  _+-eq_ : A ⇔ B → C ⇔ D → (A + C) ⇔ (B + D)
```

Given *cast* it is easy to define *resp*. It is also easy to prove that *cast* maps the identity to the identity (assuming extensionality), from which we get *resp-id*.

Some readers may wonder why we include both type and k in U: in the development above type is treated in exactly the same way as k *Type*. The reason is that we want to discuss the following variant of *Is-isomorphism*, defined recursively as a logical relation:

```
Is-isomorphism′ : ∀ a {B C} . B ≃ C → El a B → El a C → Type₁
Is-isomorphism′ id      eq = λ x y. to eq x ≡ y
Is-isomorphism′ type    eq = λ X Y. X ≃ Y
Is-isomorphism′ (k A)   eq = λ x y. x ≡ y
Is-isomorphism′ (a ⇾ b) eq = Is-isomorphism′ a eq →-rel
                             Is-isomorphism′ b eq
Is-isomorphism′ (a ⊗ b) eq = Is-isomorphism′ a eq ×-rel
                             Is-isomorphism′ b eq
Is-isomorphism′ (a ⊕ b) eq = Is-isomorphism′ a eq +-rel
                             Is-isomorphism′ b eq
```

Note that the type and k cases are not identical. The relation combinators used above are defined as follows:

```
(P →-rel Q) f        g        = ∀ x y. P x y → Q (f x) (g y)
(P ×-rel Q) (x, u)   (y, v)   = P x y × Q u v
(P +-rel Q) (inj₁ x) (inj₁ y) = P x y
(P +-rel Q) (inj₁ x) (inj₂ v) = ⊥
(P +-rel Q) (inj₂ u) (inj₁ y) = ⊥
(P +-rel Q) (inj₂ u) (inj₂ v) = Q u v
```

The definition of *Is-isomorphism′* can perhaps be seen as more natural than that of *Is-isomorphism*. However, we can prove that they are in bijective correspondence by recursion on the structure of a:

```
∀ a B C x y. (eq : B ≃ C) →
Is-isomorphism a eq x y ↔ Is-isomorphism′ a eq x y
```

We omit our proof, but note that only the type case uses univalence (the _⇾_ case uses extensionality).

3.5 Examples

Let us now consider some examples.

Monoids We can define monoids in the following way:

```
monoid : Code
monoid =
  ((id ⇾ id ⇾ id)                      -- Binary operation.
     ⊗
   id                                  -- Identity.
  , λ C (_•_, e) .
    ((Is-set C ×                       -- C is a set.
      (∀ x. e • x ≡ x) ×               -- Left identity.
      (∀ x. x • e ≡ x) ×               -- Right identity.
      (∀ x y z. x • (y • z) ≡ (x • y) • z)  -- Associativity.
     )
    , . . .  -- The laws are propositional (assuming extensionality).
    )
  )
```

Note that we require the carrier type C to be a set. We omit the proof showing that the monoid laws are propositional. The proof makes use of the fact that C is a set (which implies that C-equality is propositional); recall that, when proving that a type is propositional, one can assume that it is inhabited (see Section 2.5).

If we unfold *Instance monoid* in a suitable way, then we see that we get a proper definition of monoids on sets:

```
$\Sigma\ C\ :\ Type_1.$
$\Sigma\ (\_\bullet\_, e)\ :\ (C\ \to\ C\ \to\ C)\ \times\ C.$
$Is\text{-}set\ C\ \times$
$(\forall\ x.\ e \bullet x\ \equiv\ x)\ \times$
$(\forall\ x.\ x \bullet e\ \equiv\ x)\ \times$
$(\forall\ x\ y\ z.\ x \bullet (y \bullet z)\ \equiv\ (x \bullet y) \bullet z)$
```

Let us now assume that we have two monoids $M_1\ =\ (C_1, (\_\bullet_1\_, e_1), laws_1)$ and $M_2\ =\ (C_2, (\_\bullet_2\_, e_2), laws_2)$. *Isomorphic monoid* M_1 M_2 has the following unfolding:

```
$\Sigma\ eq\ :\ C_1\ \simeq\ C_2.$
$(\lambda\ x\ y.\ to\ eq\ (from\ eq\ x \bullet_1 from\ eq\ y)), to\ eq\ e_1)\ \equiv\ (\_\bullet_2\_, e_2)$
```

Monoid isomorphisms are typically defined as homomorphic bijections, whereas our definition states that an isomorphism is a homomorphic equivalence. However, these differences are mainly superficial: equivalences and bijections *on sets* are in bijective correspondence (assuming extensionality).

Posets Let us now define posets:

```
poset : Code
poset =
  (id ⇀ id ⇀ type                                  -- The ordering relation.
  , λ C _⩽_.
    ((Is-set C ×                                   -- C is a set.
      (∀ x y. Is-proposition (x ⩽ y)) ×            -- Pointwise
                                                   -- propositionality.
      (∀ x. x ⩽ x) ×                               -- Reflexivity.
      (∀ x y z. x ⩽ y → y ⩽ z → x ⩽ z) ×           -- Transitivity.
      (∀ x y. x ⩽ y → y ⩽ x → x ≡ y)               -- Antisymmetry.
     )
    , ...  -- The laws are propositional (assuming extensionality).
    )
  )
```

It is easy to prove that the laws are propositional by making use of the assumptions that the carrier type is a set and that the ordering relation is pointwise propositional.

Instance poset has the following unfolding:

```
$\Sigma\ C\ :\ Type_1.$
$\Sigma\ \_\leqslant\_\ :\ C\ \to\ C\ \to\ Type.$
```

$$\begin{array}{l}
\mathit{Is\text{-}set}\ C\ \times \\
(\forall\ x\ y.\ \mathit{Is\text{-}proposition}\ (x \leqslant y))\ \times \\
(\forall\ x.\ x \leqslant x)\ \times \\
(\forall\ x\ y\ z.\ x \leqslant y\ \rightarrow\ y \leqslant z\ \rightarrow\ x \leqslant z)\ \times \\
(\forall\ x\ y.\ x \leqslant y\ \rightarrow\ y \leqslant x\ \rightarrow\ x\ \equiv\ y)
\end{array}$$

For posets $P_1 = (C_1, \_{\leqslant_1}\_, \mathit{laws}_1)$ and $P_2 = (C_2, \_{\leqslant_2}\_, \mathit{laws}_2)$ we get that *Isomorphic poset* P_1 P_2 is definitionally equal to

$$\Sigma\ \mathit{eq}\ :\ C_1\ \simeq\ C_2.\ (\lambda\ a\ b.\ \mathit{from\ eq}\ a \leqslant_1 \mathit{from\ eq}\ b)\ \equiv\ \_{\leqslant_2}\_.$$

This definition is not identical to the following definition of order isomorphism:

$$\Sigma\ \mathit{eq}\ :\ C_1\ \leftrightarrow\ C_2.\ \forall\ a\ b.\ (a \leqslant_1 b)\ \Leftrightarrow\ (\mathit{to\ eq}\ a \leqslant_2 \mathit{to\ eq}\ b)$$

However, in the presence of univalence the two definitions are in bijective correspondence:

$$\begin{array}{ll}
\Sigma\ \mathit{eq}\ :\ C_1\ \simeq\ C_2.\ (\lambda\ a\ b.\ \mathit{from\ eq}\ a \leqslant_1 \mathit{from\ eq}\ b)\ \equiv\ \_{\leqslant_2}\_ & \leftrightarrow \\
\Sigma\ \mathit{eq}\ :\ C_1\ \leftrightarrow\ C_2.\ (\lambda\ a\ b.\ \mathit{from\ eq}\ a \leqslant_1 \mathit{from\ eq}\ b)\ \equiv\ \_{\leqslant_2}\_ & \leftrightarrow \\
\Sigma\ \mathit{eq}\ :\ C_1\ \leftrightarrow\ C_2.\ \forall\ a\ b.\ (\mathit{from\ eq}\ a \leqslant_1 \mathit{from\ eq}\ b)\ \equiv\ (a \leqslant_2 b) & \leftrightarrow \\
\Sigma\ \mathit{eq}\ :\ C_1\ \leftrightarrow\ C_2.\ \forall\ a\ b.\ (a \leqslant_1 b)\ \equiv\ (\mathit{to\ eq}\ a \leqslant_2 \mathit{to\ eq}\ b) & \leftrightarrow \\
\Sigma\ \mathit{eq}\ :\ C_1\ \leftrightarrow\ C_2.\ \forall\ a\ b.\ (a \leqslant_1 b)\ \simeq\ (\mathit{to\ eq}\ a \leqslant_2 \mathit{to\ eq}\ b) & \leftrightarrow \\
\Sigma\ \mathit{eq}\ :\ C_1\ \leftrightarrow\ C_2.\ \forall\ a\ b.\ (a \leqslant_1 b)\ \Leftrightarrow\ (\mathit{to\ eq}\ a \leqslant_2 \mathit{to\ eq}\ b) &
\end{array}$$

The first step uses the fact that bijections between sets are in bijective correspondence with equivalences, the second step uses the key property of equality of Π-types from Section 2.3, the third step uses the fact that *from eq* and *to eq* are inverses, the fourth step uses univalence, and finally the last step uses the fact that, for *propositions*, equivalence ($\_\simeq\_$) and logical equivalence ($\_\Leftrightarrow\_$) are in bijective correspondence. (Every step makes use of the assumption of extensionality.)

If we had used *Is-isomorphism′* (see Section 3.4) instead of *Is-isomorphism* in the definition of *Isomorphic*, then *Isomorphic poset* P_1 P_2 would have been definitionally equal to

$$\begin{array}{l}
\Sigma\ \mathit{eq}\ :\ C_1\ \simeq\ C_2. \\
\forall\ a\ b.\ \mathit{to\ eq}\ a\ \equiv\ b\ \rightarrow\ \forall\ c\ d.\ \mathit{to\ eq}\ c\ \equiv\ d\ \rightarrow\ (a \leqslant_1 c)\ \simeq\ (b \leqslant_2 d).
\end{array}$$

One can prove that this expression is in bijective correspondence with the definition of order isomorphism above without using the univalence axiom.

Discrete Fields In constructive mathematics there are several non-equivalent definitions of fields. One kind of *discrete* field consists of a commutative ring with zero distinct from one, plus a multiplicative inverse operator. We restrict attention to the specification of this operator, and choose to specify it as a partial operation:

$$\mathsf{id} \rightarrowtriangle (\mathsf{k}\ \top \oplus \mathsf{id})$$

Let us use the name $\_^{-1}$ for the operator. It should satisfy the following laws, where 0, 1 and $\_\cdot\_$ stand for the ring's zero, one and multiplication:

$$\begin{array}{llcll}
\forall\, x. & x^{-1} \equiv \mathsf{inj}_1\, \mathsf{tt} & \to & x & \equiv 0 \\
\forall\, x\; y. & x^{-1} \equiv \mathsf{inj}_2\, y & \to & x \cdot y & \equiv 1
\end{array}$$

These laws are propositional, given the other laws and extensionality, so this specification of discrete fields fits into our framework.

(We have proved that our definition of discrete fields is in bijective correspondence with non-trivial discrete fields, as defined by Bridges and Richman (1987), using $\_{\equiv}\_$ as the equality relation, and $\lambda\, x\; y.\; x \equiv y \to \bot$ as the inequality relation. In fact, Bridges and Richman's definition, restricted in this way, also fits into our framework.)

Fixpoint Operators All the examples above use first-order operators. As an example of the use of higher-order types we consider sets equipped with fixpoint operators:

```
set-with-fixpoint-operator : Code
set-with-fixpoint-operator =
  ((id ⇾ id) ⇾ id
  , λ C fix.
    ((Is-set C ×
      (∀ f. f (fix f) ≡ fix f)
     )
    , ...
    )
  )
```

Given the instances $F_1 = (C_1, \mathit{fix}_1, \mathit{laws}_1)$ and $F_2 = (C_2, \mathit{fix}_2, \mathit{laws}_2)$ we get that *Isomorphic set-with-fixpoint-operator* F_1 F_2 is definitionally equal to

$$\Sigma\; \mathit{eq} : C_1 \simeq C_2.\; (\lambda\, f.\; \mathit{to}\; \mathit{eq}\; (\mathit{fix}_1\; (\lambda\, x.\; \mathit{from}\; \mathit{eq}\; (f\; (\mathit{to}\; \mathit{eq}\; x))))) \equiv \mathit{fix}_2.$$

If we had used *Is-isomorphism′* instead of *Is-isomorphism* in the definition of *Isomorphic*, then we could have obtained the following unfolding instead:

$$\begin{array}{l}
\Sigma\; \mathit{eq} : C_1 \simeq C_2. \\
\forall\, f\; g.\; (\forall\, x\; y.\; \mathit{to}\; \mathit{eq}\; x \equiv y \to \mathit{to}\; \mathit{eq}\; (f\; x) \equiv g\; y) \to \\
\quad \mathit{to}\; \mathit{eq}\; (\mathit{fix}_1\; f) \equiv \mathit{fix}_2\; g
\end{array}$$

This unfolding is perhaps a bit easier to understand.

4 Related Work

The first use of Σ-types—or "telescopes" (de Bruijn 1991)—to formalise abstract mathematical structures is possibly due to Zucker (1977), one of the members of the AUTOMATH project team.

The notion of structure used in Section 3 (instantiated as in Section 3.4) can be seen as a type-theoretic variant of Bourbaki's notion of structure (1957), using type-theoretic function spaces instead of power sets. (In Bourbaki's setting a function is defined as a functional relation, and a relation is an element of the power set of a cartesian product.) Furthermore the notion of isomorphism that Bourbaki associates to a structure is very similar to the one used in this paper.

The main theorem in Section 3.3 can be contrasted to what happens for Bourbaki's notion of structure formulated in set theory. As observed in the introduction the membership relation can be used to distinguish between isomorphic monoids. However, it is possible to restrict attention to relations that are "transportable", i.e. relations that respect isomorphisms (Bourbaki 1957). Marshall and Chuaqui (1991) state that set-theoretical sentences are transportable iff they are equivalent (in a certain sense) to type-theoretical sentences (for certain variants of set and type theory).

The simple result that we present in this paper, a first version of which was formalised in Agda in March 2011, is only a starting point. Aczel and Shulman's "Abstract SIP Theorem" (2013), which at the time of writing is under development, is more abstract. An important point of our formalisation is that we do not assume that we have a definitional computation rule for J (as discussed in Section 2.3). Based on our experience of working without a definitional computation rule we expect that Aczel and Shulman's result can also be proved in this setting.

Aczel and Shulman (2013) also present a different kind of generalisation. We can state it as follows: in type theory extended with the axiom of univalence, and using natural definitions of "category" and "equivalence of categories", equivalence of two categories C and D is in bijective correspondence with equality of C and D.

5 Conclusions

We have shown that, for a large class of algebraic structures, isomorphism is in bijective correspondence with equality.

The results can be generalised further. For instance, the development above is restricted to a single carrier type, and uses simple types. The accompanying code contains a development with support for multiple carrier types as well as polymorphic types. However, this development uses a computing J rule. It is also more complicated, so in the interest of readability we have chosen not to present this development.

Acknowledgements

The authors acknowledge financial support from the ERC: "The research leading to these results has received funding from the European Research Council under the European Union's Seventh Framework Programme (FP7/2007-2013) / ERC grant agreement n° 247219."

References

Peter Aczel and Michael Shulman. Category theory. In *The HoTT Book*. 2013. Draft, available at `http://uf-ias-2012.wikispaces.com/The+book`.

The Agda Team. The Agda Wiki. Available at `http://wiki.portal.chalmers.se/agda/`, 2013.

N. Bourbaki. *Théorie des ensembles*, volume 1 of *Éléments de Mathématique*, chapter 4: Structures. Hermann, 1957.

Douglas Bridges and Fred Richman. *Varieties of Constructive Mathematics*, volume 97 of *London Mathematical Society Lecture Note Series.* Cambridge University Press, 1987. doi:10.1017/CBO9780511565663.

N. G. de Bruijn. Telescopic mappings in typed lambda calculus. *Information and Computation*, 91(2):189–204, 1991. doi:10.1016/0890-5401(91)90066-B.

N.G. de Bruijn. Set theory with type restrictions. In *Infinite and Finite Sets, to Paul Erdős on his 60th birthday, Vol. I*, volume 10 of *Colloquia Mathematica Societatis János Bolyai*, pages 205–214. North-Holland Publishing Company, 1975. A reprint is available (doi:10.1016/S0049-237X(08)70229-5).

N.G. de Bruijn. A survey of the project AUTOMATH. In *To H.B. Curry: Essays on Combinatory Logic, Lambda Calculus and Formalism*, pages 579–606. Academic Press, 1980. A reprint is available (doi:10.1016/S0049-237X(08) 70203-9).

Michael Hedberg. A coherence theorem for Martin-Löf's type theory. *Journal of Functional Programming*, 8(4):413–436, 1998. doi:10.1017/ S0956796898003153.

Adolf Lindenbaum and Alfred Tarski. On the limitations of the means of expression of deductive theories. In *Logic, Semantics, Metamathematics: Papers from 1923 to 1938, second edition.* Hackett Publishing Company, 1983. Translated by J. H. Woodger.

M. Victoria Marshall and Rolando Chuaqui. Sentences of type theory: The only sentences preserved under isomorphisms. *The Journal of Symbolic Logic*, 56 (3):932–948, 1991. doi:10.2307/2275062.

Per Martin-Löf. An intuitionistic theory of types: Predicative part. In *Logic Colloquium '73*, volume 80 of *Studies in Logic and the Foundations of Mathematics*, pages 73–118, 1975. doi:10.1016/S0049-237X(08)71945-1.

R.P. Nederpelt and J.H. Geuvers. Twenty-five years of Automath research. *Studies in Logic and the Foundations of Mathematics*, 133:3–54, 1994. doi:10. 1016/S0049-237X(08)70198-8.

Bengt Nordström, Kent Petersson, and Jan M. Smith. *Programming in Martin-Löf's Type Theory: An Introduction.* Oxford University Press, 1990.

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

Alfred Tarski. What are logical notions? *History and Philosophy of Logic*, 7 (2):143–154, 1986. doi:10.1080/01445348608837096. Published posthumously, edited by John Corcoran.

Vladimir Voevodsky. Univalent foundations project (a modified version of an NSF grant application). Unpublished, 2010.

Vladimir Voevodsky. Development of the univalent foundations of mathematics in Coq. Available at `https://github.com/vladimirias/Foundations/`, 2011.

J. Zucker. Formalization of classical mathematics in AUTOMATH. In *Colloque International de Logique, Clermont-Ferrand, 18-25 juillet 1975*, volume 249 of *Colloques Internationaux du Centre National de la Recherche Scientifique*, pages 135–145, 1977. A reprint is available (doi:10.1016/S0049-237X(08)70202-7).