Proving Univalence for Generic Higher Structures and Specific Monoids on Sets

Raul Santana Trejo
Supervisors: Benedikt Ahrens, Kobe Wullaert
EEMCS, Delft University of Technology, The Netherlands

June 19, 2022

A Dissertation Submitted to EEMCS faculty Delft University of Technology,
In Partial Fulfilment of the Requirements
For the Bachelor of Computer Science and Engineering

Abstract

Type Theory enables mathematicians to perform proofs in a formal language that computers can understand. This enables computer-assisted proofs and the computerization of all mathematical knowledge. Homotopy Type Theory (HoTT) views types as topological spaces, unlocking new ways to understand and expand Type Theory. One of the most interesting expansions of Type Theory thanks to HoTT has been the development of the Univalence Axiom which expands the definition of "equality between types". This paper analyses the work done in the paper "Isomorphims is Equality" and applies their proof of Univalence for higher structures to two specific monoids on sets. This is done in terminology that is understandable to Computer Scientists with the aim to further the collaboration between the two fields.

1 Introduction

Mathematics has a lot to offer to the Computer Science world by providing us models to understand and verify code without even having to run it. Type Theory is one of these tools, it allows us to construct mathematical proofs that computers can understand. This has lead to developments such as TRX, an interpreter developed in Coq that is able to produce formally correct programs [7]. Mathematicians have recently developed a very promising new branch of Type Theory called Homotopy Type Theory (HoTT), which enables mathematicians (and consequently Computer Scientists) to reason about much broader problems, and generalize findings between different mathematical structures. Much of this early research is very mathematical in nature so we find a knowledge gap in making this knowledge accessible to Computer Scientists. With this paper we attempt to translate those findings in terms that are more familiar to Computer Scientists so we can harness the power of Homotopy Type Theory in our programs and further the collaboration between the mathematics and computational worlds, like was the case with recent research that uses Homotopy Type Theory to avoid combinatorial explosion bottlenecks in Big Data [8].

One recent research paper in the HoTT field is "Isomorphism is Equality" by Coquand and Danielsson [3] (also referred to as the "reference paper" further on). This paper proofs univalence for higher structures by first creating a way to define such structures in Type Theory and then using these definitions in their proof. We will go over the proofs in the paper using two specific isomorphic monoids and relating each step of definition and proof to the specific monoids, we hope this helps in understanding the proof, especially for people unfamiliar with very abstract mathematical thinking.

To achieve this explanation we tackle the question, "How do the proofs for complex structures from "Isomorphism is Equality" translate to Computer Science terminology and how do we apply the proof to specific monoids?". This question is then divided into three subquestions:

- What is the intuition behind Homotopy Type Theory in Computer Science terms?
- What is the Univalence Axiom in Computer Science terms?
- What is the intuition behind the proofs in the "Isomorphism is equality" paper?
- How do the proofs on the paper match the process of defining univalence between two specific monoids?

We start in the Preliminary section where we explain the basic notion behind Type Theory as a basis for mathematics, what the Homotopy Type Theory interpretation is, and what the Univalence Axiom states. We also take the chance to explain the key preliminary concepts behind the proof of "Isomorphism is Equality". In the Analysis section we then make a step-by-step walk-through of the paper using different explanations and referring to two specific monoids at every step. Explaining it in this terminology helps Computer Scientists to understand how to use the Univalence Axiom for higher proofs and takes us further to applying this knowledge to structures as complex as entire programs. We finalize the paper with recommended next steps and implications of the research.

2 Preliminaries

This section introduces the reader to the background knowledge on Homotopy Type Theory necessary for understanding the proofs in this paper. The core of this knowledge comes from the Homotopy Type Theory textbook which contains alternative explanations for all presented concepts along with further details, proofs and implications [1]. This section attempts to transmit the knowledge from the book in terms that are understandable for someone with a Computer Science bachelors degree. First we explain how Type Theory is used to perform proofs in a computer-friendly manner, then we dive into Homotopy Type Theory, the main intuitions and how they lead to the reference paper.

2.1 Type Theory and Computer Science

2.1.1 How Type Theory can be used for mathematical proofs

Most computer scientists will be familiar with the concept of a Type Checker, a computer program that can check all the declarations and operations in a piece of code to ensure that all the steps are valid and create an object of the desired type. This process is very similar to the process of checking a mathematical proof, one starts with several definitions and applies them to create a new structure that satisfies the desired conclusion. We can use Type Checking algorithms to check mathematical proofs, as long as we can write our proofs in terms of Types, this is the concept of Type Theory: defining mathematics in terms of types that can be understood and checked by a Type Checker.

Mathematical proofs have always been written by hand. This has lead to big debates on the validity of some proofs such as the case of the ABC Conjecture [6], by using Type Theory and computer-checkers we can define the entirety of mathematics in a formal, indexed and verifiable way.

The key insight in Type Theory as a basis for mathematics is to view types as propositions. Any theorem, assumption and conclusion corresponds to a specific Type. We can then define other logical concepts such as implications, conditional logic and predicate logic using several other structures that are common in computer programs. A proof finally looks like a function that takes in several arguments (assumptions) and applies operations to them (logical rules) to derive a final output (conclusion). If the final output's Type is the one that corresponds to the desired conclusion, we have proven that conclusion.

2.2 Homotopy Type Theory

One interesting question that mathematicians have been diving into is "What is a type?". There are several models that help us reason about types, the Homotopy Type Theory model thinks of a type as a space, and equalities between two objects is related to the ability to go from one object to the other within the space, this is explained in detail in the following sections.

2.2.1 Types as spaces

In HoTT, types are seen as spaces, with instances of that type being points in the space. So the Type of Integers could be seen as a space with several disconnected points, each point being a specific Integer. Other logical notions have a topological equivalent, for example two objects are equal if there is a path between them, a function from $A \to B$ transforms space A into space B by assigning every point of A a corresponding point in B. These are also shown in the table below.

Computer Science	**Types as spaces**
$Integer\ i$	i is a point in the space of all $Integers$
$Equality : a == b$	there is a path between a and b
$f : Integer \to \mathbf{N}$	f maps the space of Integers to the space of Natural Numbers

2.2.2 Equality between spaces and the Univalence Axiom

One of the key questions in Homotopy Type Theory is the concept of equality. When are two objects equal? Traditional definitions of equality as seen in Set Theory are often too strict, same applies with traditional equality between spaces. For example take the two paths in Figure 1.

Figure 1: Two similar paths between two points

It would be naive to consider them completely distinct just because they are not point-wise equal. In Homotopy Type Theory two paths are considered equal if there is a continuous transformation between them as shown in Figure 2.

The rise of Homotopy Type Theory came from Vladimir Voevodsky when he realized this concept of "equality coming from transformations" could be applied to entire spaces, and thus entire Types. This concept is called the "Univalence Axiom" (UA). The UA states that two objects that have an Isomorphism between them are equal. An Isomorphism is a continuous transformation back and forth (like the paths in figure 2 above). So the

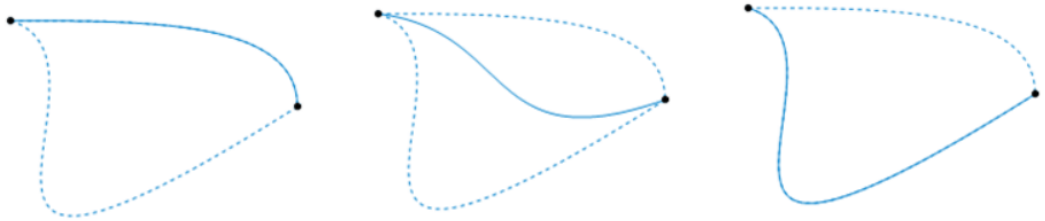

Figure 2: Continuous transformation between two paths

Univalence Axiom states that we can define equality between entire Types as the condition that there exists a continuous transformation back and forth between them. Isomorphism is the same as equality. When applying the univalence axiom we will use the function $\simeq \Longrightarrow \equiv$, this name comes from transforming an equivalence ($\simeq$) into an equality ($\equiv$).

2.2.3 Higher Structures

The univalence axiom applies to Types that form basic spaces, but does not necessary apply to Types that have additional conditions (such as monoids, discrete fields, or posets). So the Univalence Axiom applies to the Natural numbers, but does not immediately apply to two monoids building on the Natural numbers. In this paper we call these advanced types that have additional conditions "Higher Structures" and we will see later on that as long as we can define our Higher Structures in certain ways within Type Theory, we can proof univalence axiom holds for them.

2.3 Monoids

We will particularly look into the example of monoids as the Higher Structure of choice. A monoid is a mathematical structure composed of three elements: a Carrier Type, a binary operator and an identity element. The Carrier Type states what type of objects the monoid works with, the binary operator is an operation that combines two elements of the Carrier Type, and the identity element is a special element of the Carrier Type such that when combined with another element n using the binary operator, the output will always be n. For example we have the monoid $(\mathbf{N}, \lambda mn.m + n, 0)$ with Carrier Type $\mathbf{N}$, binary operator $(\lambda mn.m + n)$ and identity element 0. Monoids have three rules, left identity, right identity and associativity. Left identity states that the "identity on the binary operator" works when the identity element is the first argument, right identity states that identity holds when the identity element is the second argument, and associativity says that concatenating the binary operation in different orders has no effect on the result $(a + b) + c \equiv a + (b + c)$. We will encode all of the above information in Type Theory in order to proof Univalence for monoids.

Monoids are highers structures because they start with a basic Type (Carrier Type) and add extra structure and rules on them (the binary operator, identity element and monoid rules).

2.3.1 Dependent Pairs

Dependent pairs are a Type-theory concept that is heavily used in the proof we discuss below. A dependent pair is a pair of items where the Type of the second element depends on the first. Dependent Pairs can be used as the Type Theory version of "there-exists" statements from predicate logic. "There exists x such that P" can be seen as a pair $(x, P\ x)$ where the existence of x is necessary to construct P. In the sections below we denote dependent pairs as $\Sigma\ x.\ P\ x$, and if we want to specify the type of x we use $\Sigma\ x : X.\ P\ x$, (there exists an x of type X such that $P\ x$ holds).

2.3.2 resp, subst and the transport theorem

One of the key advantages of working with equalities and similarities is the ability to translate findings from one object to the other. For example in traditional algebra if we know $x = y$, we can substitute all instances of x by instances of y, which may lead to interesting findings, proofs or simplifications.

Let P be some operation on a space and let A and B be two spaces. An interesting case is when P "respects isomorphism", essentially meaning that $A \simeq B \to P\ A \to P\ B$, we call this property *resp*, it means that P transforms the space of A in such a way that we can still transform from $P\ A$ to $P\ B$. More specifically $resp\ eq\ (PA) : P\ B$, we transform $P\ A$ into $P\ B$ using the original isomorphism eq. This allows us to make claims about B thanks to the fact that it is isomorphic to A.

Another useful thing to do with equalities is substitution, we define substitution as $subst : x \equiv y \to P\ x \to P\ y$, essentially if $x \equiv y$ and $P\ x$ holds, then $P\ y$ must hold. We can substitute equal objects in any proposition P.

Notice that the univalence axiom allows us to transform equivalences to equalities, this means that *subst* also applies to equivalences, if we have $eq : A \simeq B$, we can transform it to an equality $eq : A \equiv B$ using univalence. More specifically, given a proof p of $P\ A$ we have $resp\ eq\ p \equiv subst\ (\simeq \Longrightarrow \equiv eq)\ p$, with both sides being a proof $P\ B$. The fact that A and B are isomorphic implies that we can substitute any instance of $P\ A$ by an instance of PB. This is called the "Transport Theorem", we can substitute two isomorphic objects.

3 Analysis

The paper "Isomorphism is equality" begins with two monoids $(\mathbf{N}, \lambda mn.m + n, 0)$ and $(\mathbf{N} \setminus 0, \lambda mn.m + n - 1, 1)$. These monoids are isomorphic but are not equal in traditional set theory. In this section we will proof these two equal under HoTT, following the steps of the reference paper and relating each step to these two original monoids.

From now on we will call these monoids $M1$ and $M2$ respectively, and for conciseness we assign each component of the monoid a different variable name, that is:

$$\begin{aligned} M1 &= (\mathbf{N}, \lambda mn.m + n, 0) = (C_1, b_1, e_1) \\ M2 &= (\mathbf{N} \setminus \{0\}, \lambda mn.m + n - 1, 1) = (C_2, b_2, e_2) \end{aligned} \tag{1}$$

For example we write b_1 to refer to $M1$'s binary operation, $\lambda mn.m + n$. These monoids are isomorphic as witnessed by isomorphism $\lambda n.n + 1$, we will call this isomorphism eq_M and use it later on to proof $M1 \equiv M2$.

3.1 Defining higher structures and similarity in Type Theory

The first step is defining the Higher Structures as explained in Section 2.2.3 using Type Theoretic terms. First we give a generic approach to defining higher structures (how we describe what a "monoid" is), then we define what an instance of a structure is composed of (how we describe specific monoids $M1$ and $M2$), and finally we create a definition of equality and isomorphism (What does $M1 \equiv M2$ and $M1 \simeq M2$ even mean).

3.1.1 The universe of all higher-structures

First we build a universe that encompases all higher structures U, and define a function $El : U \to Type \to Type$. El is a function that takes in a specific higher-structure $a : U$ and represents all instances of that higher-structure. For example $El\ monoid\ \mathbf{N}$ describes all monoids that have the Natural Numbers as their carrier set, so $M1$ is an element of type $El\ monoid\ \mathbf{N}$ while $M2$ is an element of $El\ monoid\ \mathbf{N}\backslash\{0\}$.

This El is a transformation of the Carrier Type, it expands the Carrier Type into the more complex higher-structure. It is important later on that El respects Isomorphism, meaning that if two Carrier Types B and C are isomorphic then $El\ a\ B$ and $El\ a\ C$ are also isomorphic, this property is the $resp$ property that we defined in section 2.3.2, so:

$$resp\ :\ \forall\ a\ B\ C.\ B \simeq C \to El\ a\ B \to El\ a\ C \tag{2}$$

Going back to our original $M1$ and $M2$, $resp$ states that since C_1 and C_2 are isomorphic through eq_N, we can transport any instance of $El\ a\ C_1$ to an instance of $El\ a\ C_2$ by moving along eq_M, we can morph the space $M1$ into the space of $M2$. According to the transport theorem as mentioned in section 2.3.2, this means we can substitute $M1$ by $M2$ in any proposition thanks to eq_M.

Now that we have defined a universe that can host our higher-structures, let's use it to give generic definitions for higher-structures.

3.1.2 Defining generic and specific higher structures

We define Higher Structures using Codes and Instances. A $Code$ is a generic description of a structure, for example we could have a $Code$ for "monoids". A $Code$ consists of a (the structure's features) and P (a family of propositions). a is the special components of the structure (like the binary operator and identity elements of monoids), and P represents the rules for the structures, such as the monoid laws (identity and associativity).

An $Instance$ is a specific formulation of a structure, it turns a given $Code$ into a C: the carrier type, x: the structure components (in other words a value of type $El\ a\ C$), and p: a proof that x satisfies all the $Code$'s propositions. To reiterate we have:

$$\begin{aligned} &Code = (a, P) \\ &Instance\ (a, P) = (C, x, p) \end{aligned} \tag{3}$$

Going back to our monoids we can generally describe them as follows:

$$\begin{aligned} &monoid\ :\ Code \\ &monoid = ((b, e), laws) \end{aligned} \tag{4}$$

Where b is a binary operator on the Carrier Type, e is an element of the Carrier Type and *laws* is a function that constructs the monoid laws from the given Carrier Type, b and e.

More specifically we define:

$$
\begin{aligned}
&b = (id \to id \to id)\\
&e = id\\
&laws = \lambda C(\_ \bullet \_, e).\\
&\quad ((Is - Set\ C)\times\\
&\quad (\forall x.e \bullet x \equiv x)\times\\
&\quad (\forall x.x \bullet e \equiv x)\times\\
&\quad (\forall x\ y\ z.x \bullet (y \bullet z) \equiv (x \bullet y) \bullet z)
\end{aligned}
\tag{5}
$$

b and e just contain the given arguments belonging to the carrier type. The *laws* object is more interesting. It first gives name to the arguments by using a λ expression, specifically it defines the Carrier Type to be C and the pair of binary operator and identity element to be $\_ \bullet \_$ and e respectively. Defining the binary operator as $\_ \bullet \_$ allows us to place arguments in the underscores. Once the arguments have been named we describe how they should behave, first of all we say that the Carrier Type C should be a Set using some $Is-Set$ function (we omit this definition but it can be found in the reference paper [3]). Note that this requirement for C to be a Set is only necessary because we are restricting our work to monoids whose Carrier Types are Sets, if we wanted to expand the work on the reference paper to non-set monoids we would remove this requirement, but this leads to complications as mentioned later in section 3.2.3. After Is-$Set\ C$ we define the three main monodic laws: left identity, right identity and associativity. The left identity term $(\forall x.e \bullet x \equiv x)$ states that for any argument x, if the first argument of the operator is the identity element e then the result must just be x. The definition for right identity is similar. Finally associativity is defined for any combination of three arguments.

Basically $monoid : Code$ says that in order to define a monoid we will need three things: a binary operator, an identity element and a way to construct elements of each of the monoid laws from the given operation and element. Remember that in Type Theory constructing an element of a type means we are proving the proposition related to the type, so constructing an element of the law from the operator and identity is the same as proving the laws hold under the given operator and identity, thus they are valid operator and identity, and form a valid monoid.

An *Instance* of a monoid is just a triplet that satisfies the Code above, $M1$ and $M2$ can easily be turned into these triplets, $M1_{Instance} = (\mathbf{N}, (\lambda mn.m + n, 0), proof_{M1})$ and $M2_{Instance} = (\mathbf{N} \setminus \{0\}, (\lambda mn.m + n - 1), proof_{M2})$. We don't construct the proofs for monoid laws in this paper but they do hold.

3.1.3 Defining isomorphism and equality

The univalence axiom shows a relation between equality and isomorphism, so we need to define these two relationships for *Instances*. An *Instance* is composed of three things, a Carrier Type, an element and a proof. Isomorphism only focuses on the Carrier Type and element, since the proofs don't contribute to the algebraic properties of the structure. First we want to ensure that a given equivalence *eq* is an isomorphism for the elements of the

Instances. We do this by transforming one element x into another y using the *resp* function defined before, if this is possible then the given equivalence works for the elements.

$$\begin{aligned} &Is\text{-}isomorphism : \forall a\{B\ C\}.B \simeq C \to El\ a\ B \to El\ a\ C \to Type \\ &Is\text{-}isomorphism\ a\ eq\ x\ y\ =\ resp\ a\ eq\ x \equiv y \end{aligned} \tag{6}$$

We then want to state that *Instances* are Isomorphic when there exists an equivalence that can transform both between Carrier Types and between elements:

$$\begin{aligned} &Isomorphic : \forall c.\ Instance\ c \to Instance\ c \to Type \\ &Isomorphic\ (a, P)\ (C, x, p)\ (D, y, q) = \Sigma eq : C \simeq D.Is\text{-}isomorphism\ a\ eq\ x\ y \end{aligned} \tag{7}$$

In this case *Isomorphic* takes in a specific code (a, P) and two instances of that code (C, x, p) and (D, y, q) and sets the requirement that there exists an equivalence eq that allows us to transform between Carrier Types C and D, and also allows us to transform between instances x and y. Since we can transform between the two key components of the *Instances* we can transform between *Instances* and thus have an Isomorphism.

Taking it to $M1$ and $M2$ using isomorphism eq_M we see that eq_M transforms between C_1 and C_2, so we satisfy the part of $\Sigma\ eq : C \simeq D$, now we need to ensure that it is also an isomorphism for two elements, so we need to show $Is\text{-}isomorphism\ a\ eq\ x\ y$. This holds because we defined in section 3.1.1 that *El* respects isomorphisms, and C_1 and C_2 are isomorphic.

Now that we have defined *Isomorphism* for *Instances* we define equality. Equality between *Instances* is equivalent to a pair containing an equality for the Carrier Types and an equality for the elements. Note that unlike in Isomorphisms, the equality function doesn't have to be the same for both Carrier Type and element, we can use two separate equalities. Essentially two *Instacnes* are equal if the Carrier Types are equal and the elements are equal, this is called the *equality-pair-lemma*.

3.2 Univalence for higher structures

3.2.1 Main Theorem

Now that we have defined Isomorphism and Equality between generic higher structures we can provide a proof that these to relationships are equal in Homotopy Type Theory under the univalence axiom. The claim has the following definition:

isomorphism-is-equality : $\forall\ a\ X\ Y\ .\ Isomorphic\ a\ X\ Y\ \leftrightarrow (X \equiv Y)$

In other words: for any two *Instances* X and Y isomorphism is equivalent to equality.

The proof in the paper is as follows:

$$\begin{aligned} &(1)\ Isomorphic\ a\ X\ Y\ \leftrightarrow \\ &(2)\ \Sigma eq : C \simeq D.\ resp\ a\ eq\ x\ \equiv y \leftrightarrow \\ &(3)\ \Sigma eq : C \simeq D.\ subst\ (El\ a)\ (\simeq \Longrightarrow \equiv eq)\ x \equiv y \leftrightarrow \\ &(4)\ \Sigma eq : C \equiv D.\ subst\ (El\ a)\ eq\ x \equiv y \leftrightarrow \\ &(5)\ X \equiv Y \end{aligned} \tag{8}$$

We start in step (1) with the assumption that we have two $Isomorphic\ Instances$ (X and Y) of some structure a and then build our way to equality from them. The transformations at each step are all bijective, so we prove the relationship both ways.

In step (2) we unfold the definition of $Isomorphic$ and state that there must exist some function eq that allows us to transform C into D and transform element x into y.

In step (3) we use the transport theorem, stating that since we can transform x into y, we can just substitute x and y, this uses the univalence axiom as explained in section 2.3.2.

Equality of entire instances needs both an equality of carrier types and an equality of elements, while eq is only an equivalence. This is where the univalence axiom is crucial, univalence states that equivalence and equality are the same thing, thus we can just assume that eq is also an equality between both the Carrier Type and elements, this results in step (4).

Finally according to the *equality-pair-lemma* the existence of an equality eq that equates the Carrier Types $C \equiv D$ and also equates the $Instances\ x \equiv y$ (by using subst on x), means these entire $Instances$ are equal, meaning we can turn step (4) into step (5), reaching our desired conclusion QED.

3.2.2 Proving equality for the two specific isomorphic monoids

We can finally proof $M1$ and $M2$ to be equal.

We start with the fact that they are $Isomorphic$, there exists an eq that can transform $\mathbf{N}$ into $\mathbf{N} \setminus \{0\}$, for example $eq_M = (\lambda n.n+1)$ satisfies this. The elements of Instances $M1$ and $M2$ are $(\lambda mn.m+n, 0)$ and $(\lambda mn.m+n-1, 1)$ respectively. Notice that eq_M also allows us to transform these elements, the outputs of $b2$ can be turned into the outputs of $b1$ using eq_M, and the $b1$ into $b2$ using the inverse $eq_M^{-1} = \lambda n.n-1$. The identity elements are also converted between eachother using eq_M and eq_M^{-1}, $eq_M 0 = 1$ and $eq_M^{-1} 1 = 0$. So we have proven that eq_M is an isomorphism between the elements, since $(eq_M\ b1, eq_M^{-1}\ e1) = (b2, e2)$. We have established that eq_M does indeed satisfy the conditions for step (2). In step (3) we show that, according to the transport theorem, we can substitute $(\lambda mn.m+n, 0)$ for $(\lambda mn.m+n-1, 1)$ by using eq_M to change from one to the other, this makes them equal. Now since we can transform between $\mathbf{N}$ and $\mathbf{N} \setminus \{0\}$ we can directly use univalence on the Carrier Types to state that $\mathbf{N} \equiv \mathbf{N} \setminus \{0\}$. We have shown that both the Carrier Types are equal ($\mathbf{N} \equiv \mathbf{N} \setminus \{0\}$) and that the instances are equal ($(\lambda mn.m+n, 0) \equiv (\lambda mn.m+n-1, 1)$) so according to the *equality-pair-lemma*, $M1$ and $M2$ are equal, QED.

3.2.3 Requirement that the monoids are built on sets

One of the key limitations of the work on monoids is the requirement that the monoid's Carrier type is a set. This requirement is used in the transition from step (3) to step (4), we are allowed to assume that the isomorphism $eq : C \simeq D$ is also an equality $eq : C \equiv D$. This transformation requires that the types of C and D are also univalent, so we can transform $\simeq \implies \equiv$, sets satisfy this requirement. Another advantage of using sets is that, as stated in the reference paper, when defining "isomorphism between monoids" we often assume a homomorphic bijection, so a relation of type $M1 \leftrightarrow M2$, while our definition wants us to start with a homomorphic equivalence $M1 \simeq M2$. When using sets these two relations are equivalent, so we are allowed to forgo the step of transforming $eq : M1 \leftrightarrow M2$ into $eq : M1 \simeq M2$.

3.3 Expanding the monoid structure

As a final observation we expand the structure we defined in section 3.1.2 to see how the system in the reference paper adapts to sets with multiple monoid structures (sets built using monoids that have more than one operator that satisfy the monoid laws). A generic definition for a set with two monoid structures could look like this:

$$
\begin{aligned}
&b = ((id \to id \to id) \times (id \to id \to id)) \\
&e = (id \times id) \\
&laws = \lambda C(\_ \bullet \_, e_\bullet, \_ \triangle \_, e_\triangle). \\
&\quad ((Is - Set\ C)\times \\
&\quad (\forall x.e_\bullet \bullet x \equiv x)\times \\
&\quad (\forall x.x \bullet e_\bullet \equiv x)\times \\
&\quad (\forall x\ y\ z.x \bullet (y \bullet z) \equiv (x \bullet y) \bullet z)\times \\
&\quad (\forall x.e_\triangle \triangle x \equiv x)\times \\
&\quad (\forall x.x \triangle e_\triangle \equiv x)\times \\
&\quad (\forall x\ y\ z.x \triangle (y \triangle z) \equiv (x \triangle y) \triangle z)
\end{aligned}
\tag{9}
$$

We define an isomorphism between two instances of these double-monoids $M_2 1 \simeq M_2 2$ as a function that can transform between the carrier sets, but also between both operators. Such a transformation respects steps (2) and (3) of our proof, since it means we can still transform the instance of the monoids, and since we define the additional operator as a tuple we preserve univalent properties, we can just create the transformed operator-tuple by applying the isomorphism to the projections of the binary operators.

In fact we can see that we can add an arbitrary number of operators to the monoid as long as we can guarantee that the single original isomorphism works for all of the new operators. To add an operator it suffices to add an operator to b and then add all the desired laws, as long as the laws are propositional. To perform the proof we just have to appropriately project the corresponding operator before applying the equivalence.

4 Responsible Research

Most mathematical research does not expose itself to the ethical impacts that other research fields have to deal with, most of the research is about discovering new ways to understand abstract concepts and the world around us. With that said there is certainly some room for the philosophical discussion of the effects that developing Type Theory will have on creative thinking, mathematics and computer science as a whole.

Type Theory takes us further into a world where everything is formally defined, but abstract concepts and conversation hold a lot of power, sometimes it is worth expressing ideas that are not perfectly described, or even complete. If we enforce a world were mathematical thinking is only acceptable once completely defined, we put a lot more pressure on the thinker and limit the room for creativity, collaboration and thinking outside the box. Computerization is generally idealized, but it may take us away from the connection between philosophy and mathematics that has always benefited both fields.

From the mathematical point of view, one must ask if Type Theory is truly the best foundation for mathematics, or if we are encouraged to think so because of the convenience of being able to work with computers while using it. The concept of a Set is more concrete

and approachable (at least at first encounter) than the abstract concept of a Type. If we choose to make the monumental effort of translating large volumes of mathematics into a new foundation we should objectively consider why we have chosen such a foundation. One valid argument is that once computerized (in any foundation) one can build machine translators that automatically translate from one computer-friendly foundation to another, but if there is a different foundation that is more computer-friendly, human-friendly and aligns better with previous mathematical work, it should be considered as an alternative to Type Theory.

In Computer Science one has to consider the implications of computers engaging in mathematics. Mathematics has been the foundation for some of the most important programs in history, from Euclid's algorithm to finding the Greatest Common Divisor of a number[9], to Fourier transforms used in signal processing [2]. Mathematics is a powerful tool that can have implications beyond our initial impression. To allow computers to understand and create mathematical proofs, is to allow them to create all sorts of algorithms that we do not understand. These algorithms could have negative effects on the areas of cryptography, data processing and statistical analysis, we should consider how much freedom and trust we place into computer-generated proofs and algorithms.

5 Conclusions and Future Work

The above is an application of the findings in "Isomorphism is Equality" [3] to two specific monoids on sets, in an attempt to explain the contents of the paper in different terms and show how the findings in the paper apply to real instances. The definitions of *Code* and *Instance* in the paper work really well for the case of monoids, it is very easy to translate a monoid into these definitions, and the generic proofs of the paper match naturally with the specific monoid instances we chose. Further research could be done in testing the specific instances of the other structures in the paper: Discrete Fields and Posets. Another avenue of research would be proving Univalence for other types of Higher Structures, such as Monoids that are not built on Sets or groupoids, such work includes research on univalent typoids [10]. Ultimately it will be fascinating to see how univalence can be implemented into existing computer programs, since this will further the development of formally correct programs and languages. This includes works such as the Coq framework implemented by Tabareau et al. that "allows the user to transparently transfer definitions and theorems for a type to an equivalent one, as if they were equal. For instance, this makes it possible to conveniently switch between an easy-to-reason-about representation and a computationally efficient representation as soon as they are proven equivalent." [5]. Another approach is developing compilers for new type theories and exploring their applications, such as recent work by Abel et al. on a cubical type theory compiler [4]

References

[1] *Homotopy type theory: Univalent foundations of mathematics*. Institute for Advanced Study, 2013.

[2] Ronald Newbold Bracewell. *The Fourier transform and its applications*, volume 31999. McGraw-hill New York, 1986.

[3] Thierry Coquand and Nils Anders Danielsson. Isomorphism is equality. *Indagationes Mathematicae*, 24(4):1105–1120, 2013. In memory of N.G. (Dick) de Bruijn (1918 - 2012).

[4] Andreas Abel et al. Compiling programs with erased univalence. 2021.

[5] Nicolas Tabareau et al. The marriage of univalence and parametricity. *J. ACM*, 68(1), jan 2021.

[6] Erica Klarreich. *Titans of Mathematics Clash Over Epic Proof of ABC Conjecture.*

[7] Adam Koprowski and Henri Binsztok. TRX: A formally verified parser interpreter. *Logical Methods in Computer Science*, 7(2), jun 2011.

[8] Tosiyasu L. Kunii and Masaki Hilaga. Homotopy type theory for big data. In *2015 International Conference on Cyberworlds (CW)*, pages 204–209, 2015.

[9] Ben Lynn. *Euclid's algorithm.*

[10] Iosif Petrakis. Univalent typoids, 2022.