Appendix A

Coq Project

Universal algebra homomorphisms and isomorphisms in HoTT

Andreas Aagaard Lynge
201710283

16th April 2019

1 Introduction

In this report I present the beginnings of a port of the Math Classes library [1] to the Homotopy Type Theory (HoTT) library [2] for the Coq proof assistant. The Math Classes library is developed by B. Spitters and E. van der Weegen as a basis for constructive analysis in Coq. The focus of the development in this report has been on porting the Universal Algebra parts of Math Classes to HoTT. The Coq formalisation of this can be found at `https://github.com/andreaslyn/hott-classes/tree/handin`.

The reader is assumed familiar with HoTT [3] and the Coq HoTT library [2]. Knowledge of Universal Algebra is not required, but to appreciate the results, some Universal Algebra background is useful.

Since this is a Coq project I will use a pseudo code notation close to the Coq UTF-8 syntax. The notation `x ≡ y` will denote `x` is judgmentally equal to `y` and `x = y` is the path type.

Section 2 presents a non-empty list data type used in later sections. Section 3 defines what is meant by an algebra and other basic notions in Universal Algebra. This corresponds to the file `interfaces/ua_algebra.v` in the formalisation. Section 4 introduces homomorphisms and isomorphisms and section 5 contains a proof of the main theorem in this report:

If there is an isomorphism between two algebras `A` and `B` then `A = B`.

The sections 4 and 5 correspond to the file `theory/ua_homomorphism.v` in the formalisation. Apart from a few results, the report is devoted to the proof of the above statement. All preliminary results used in the proof are given in the report or can be found in the HoTT book [3]. Section 6 concludes and compares the main theorem of this report to a similar theorem by T. Coquand and N. A. Danielsson [4].

2 Non-empty List

This section introduces a non-empty list implementation with accompanying notation used in the following sections.

Definition 2.1. A non-empty list is defined by

```
Inductive ne_list (T : Type) : Type :=
  | one : T → ne_list T
  | cons : T → ne_list T → ne_list T.
Arguments one {T}.
Arguments cons {T}.
```

For `ne_lists` we introduce the notation

```
Global Notation "[: x :]" := (one x) : ne_list_scope.
Global Notation "[: x ; .. ; y ; z :]"
    := (cons x .. (cons y (one z)) ..) : ne_list_scope.
Global Infix ":::"
    := cons (at level 60, right associativity) : ne_list_scope.
```
◊

The induction principle for the non-empty list is similar to that of the regular list. As an example, suppose `w : ne_list T` is a non-empty list and `P : ne_list T → Type` some predicate. To prove `P w` by induction we consider the base case `w ≡ [:x:]` and show that `P [:x:]` holds. Then, for the inductive step `w ≡ x ::: w'`, we assume `P w'` and show it implies `P (x ::: w')`.

3 Universal Algebra

In this section we develop the central definitions in universal algebra and provide a couple of useful results.

Definition 3.1. A *signature* is defined by

```
Record Signature : Type := BuildSignature
  { Sort : Type
  ; Symbol : Type
  ; symbol_types : Symbol → ne_list Sort }.
Definition SymbolType (σ : Signature) := ne_list (Sort σ).
```
◊

The intuition for this definition is that a signature specifies which operations (functions) an algebra for the signature is expected to provide.

- An algebra for `σ : Signature` provides a type for each *sort* `s : Sort σ`.
- The type `Symbol σ` consists of *function symbols*. For each function symbol `u : Symbol σ`, an algebra for the signature provides a corresponding operation.
- The field `symbol_types σ u` indicates which type the operation corresponding to `u` should to have.

Definition 3.2. We introduce the implicit coercion

```
Global Coercion symbol_types : Signature >-> Funclass.
```
◊

So with `σ : Signature` and `u : Symbol σ`, then `σ u ≡ symbol_types σ u` definitionally.

The `Operation` function

```
Operation : ∀ {σ : Signature}, (Sort σ → Type) → SymbolType σ → Type
```

is used to convert `σ u ≡ symbol_types σ u` into the type that the corresponding algebra operation to `u` should have.

Definition 3.3. For `A : Sort σ → Type` and `w : SymbolType σ` a symbol type,

`Operation A w := A` s_1 `→ A` s_2 `→ ··· → A` s_n `→ A t`

when `w ≡ [:`s_1`;` s_2`; ...;` s_n`; t:]` for s_1 s_2 $\cdots$ s_n `t : Sort σ`. ◊

Lemma 3.4. If `A s` is an `n`-type for all `s : Sort σ`, then `Operation A w` is an `n`-type for any `w : SymbolType σ`. In particular, if `A s` is a set for all `s`, then `Operation A w` is a set.

Proof. Induction on `w` and Theorem 7.1.9 in the HoTT book [3]. □

Definition 3.5. An *algebra* is defined by

```
Record Algebra {σ : Signature} : Type := BuildAlgebra
  { carriers : Sort σ → Type
  ; operations : ∀ (u : Symbol σ), Operation carriers (σ u)
  ; hset_carriers_algebra : ∀ (s : Sort σ), IsHSet (carriers s) }.
Arguments Algebra : clear implicits.
Arguments BuildAlgebra {σ} carriers operations {hset_carriers_algebra}.
```
◊

So an algebra `A : Algebra` σ for a signature σ consists of a type `carriers A s` for each sort `s : Sort` σ, and an *operation* `operations A u : Operation (carriers A)` (σ `u`) for each function symbol `u : Symbol` σ. Further, there is an associated proof that `carriers A s` is a set for any `s : Sort` σ.

The following lemma has the same role as the `equality-pair-lemma` by T. Coquand and N. A. Danielsson [4].

Lemma 3.6. Given two algebras `A B : Algebra` σ for a signature σ. To find a path `A = B`, it suffices to find paths between the carriers `p : carriers A = carriers B` and the operations `q : p#(operations A) = operations B`, where `p#` is transport along `p`,

```
p#(operations A) ≡ transport (λ C, ∀ u, Operation C (σ u)) p (operations A)
```

Proof. Assume we are given the above paths `p` and `q` between carriers and operations. Records are Σ-types. So to find a path `A = B`, by Theorem 2.7.2 in the HoTT book (twice), it is sufficient to find paths of type

(i) `carriers A = carriers B`,
(ii) `pr1 (p#(operations A; hset_carriers_algebra A)) = operations B`,
(iii) `pr2 (p#(operations A; hset_carriers_algebra A)) = hset_carriers_algebra B`,

where `pr1` and `pr2` are the Σ-projections. A path of type (i) is given by `p`. By path induction on `p` and the computation rules for `transport` and `pr1` there is a path of type

```
pr1 (p#(operations A; hset_carriers_algebra A)) = p#(operations A).
```

Concatenating this path with `q : p#(operations A) = operations B` we obtain a path of type (ii). It remains to find path (iii). According to Lemma 3.3.5 and Example 3.6.2 in the HoTT book [3], `(∀ s, IsHSet (carriers B s))` is a mere proposition. Thus the path type in (iii) is contractible by Lemma 3.11.10 [3], so such a path exists. □

Definition 3.7.

```
Global Coercion carriers : Algebra >-> Funclass.
Global Notation "u ^^ A" := (operations A u) (at level 60, no associativity)
                            : Algebra_scope.
```
◊

Using the above implicit coercion with `A : Algebra` σ and `s : Sort` σ, we have `A s` $\equiv$ `carriers A s` by definition.

4 Homomorphisms and isomorphisms

This section defines homomorphism and isomorphism. Then we provide some results about homomorphisms and isomorphisms. In the end some elementary homomorphisms are defined. Throughout the section we let `A B : Algebra` σ denote two algebras for a signature σ `: Signature`.

Definition 4.1. Let `f : (∀ (s : Sort σ), A s → B s)` be a family of functions. Suppose `α : Operation A w` and `β : Operation B w` are operations of types given by `w`, see Definition 3.3. We define `OpPreserving f α β : Type` to be the type:

For all x_1 `: A` s_1, x_2 `: A` s_2, …, x_n `: A` s_n,

`f t (α` x_1 x_2 ⋯ x_n`) = β (f` s_1 x_1`) (f` s_2 x_2`)` ⋯ `(f` s_n x_n`)`,

where `[:`s_1`;` s_2`; …;` s_n`; t:]` ≡ `σ u` is the symbol type of `u`.

We define *homomorphism* by

```
Record Homomorphism {σ} {A B : Algebra σ} : Type
  := BuildHomomorphism
     { def_hom : ∀ (s : Sort σ), A s → B s
     ; is_hom : ∀ (u : Symbol σ) OpPreserving def_hom (u^^A) (u^^B) }.
Arguments Homomorphism {σ}.
```

We add an implicit coercion

```
Global Coercion def_hom : Homomorphism >-> Funclass.
```

◊

With the above implicit coercion we can apply a homomorphism without using `def_hom` explicitly. We will make use of this right away:

Lemma 4.2. If `f g : Homomorphism A B` are two homomorphisms and there is a family of homotopies `p : (∀ (s : Sort σ), f s ∼ g s)`. Then `f = g`.

Proof. This is because `OpPreserving h (u^^A) (u^^B)` is a mere proposition for any `h : ∀ s, A s → B s`. See the formalisation for details. □

Definition 4.3. For `f : Homomorphism A B` a homomorphism, `IsIsomorphism f : Type` is defined as the type:

For all `s : Sort σ`, `f s` is both a surjection and an injection.

By a *surjection* we mean Definition 4.6.1(i) in HoTT [3] and by *injection* we mean:

`∀ (x y : A s), f s x = f s y → x = y.`

Since `B s` is a set, by equation (4.6.2) in the HoTT book, being an injection is equivalent to being an *embedding*, defined in Definition 4.6.1(ii) in the HoTT book.

We say that `f` is an *isomorphism* if `IsIsomorphism f` holds. ◊

Lemma 4.4. `IsIsomorphism f` is a mere proposition.

Proof. This follows from surjection and injection being mere propositions. See the formalisation. □

Lemma 4.5. Assume `f : Homomorphism A B`. If `IsIsomorphism f` then `f : (∀ s, A s → B s)` is a family of equivalences

`f : ∀ s, A s ≃ B s`

Proof. Let `s : Sort σ`. Since `f s` is both a surjection and an embedding it follows from Theorem 4.6.3 in HoTT [3] that `A s ≃ B s`. □

For the rest of this section we introduce some elementary homomorphisms and isomorphisms. We omit the proofs of `OpPreserving` and `IsIsomorphism`, which can be found in the formalisation.

Lemma 4.6. There is an *identity* homomorphism `hom_id` induced from the family of identity functions,

$$\lambda\ \texttt{(s : Sort}\ \sigma\texttt{) (x : A s), x.}$$

The identity homomorphism is an isomorphism `IsIsomorphism hom_id`. □

Lemma 4.7. Suppose `f : Homomorphism A B` and `IsIsomorphism f`. Equivalences have inverse functions, so by Lemma 4.5 there is a family of inverse functions

$$\lambda\ \texttt{(s : Sort}\ \sigma\texttt{), (f s)}^{-1}\texttt{.}$$

This family of functions gives rise to a homomorphism `hom_inv`, which is also an isomorphism `IsIsomorphism hom_inv`. This homomorphism is also referred to as the *inverse* homomorphism of `f`. □

Lemma 4.8. With `g : Homomorphism B C` and `f : Homomorphism A B` there is a *composition* homomorphism `hom_compose` with family of functions

$$\lambda\ \texttt{(s : Sort}\ \sigma\texttt{), g s}\circ\texttt{f s.}$$

If both `g` and `f` are isomorphisms then `hom_compose` is an isomorphism as well. □

5 Isomorphism is equality

This section proves the main theorem in this report. If `A B : Algebra` σ are two algebras for a signature σ and there is an isomorphism `Homomorphism A B`, then there exists a path `A = B`.

5.1 Preliminary results

We begin with `path_forall_recr_beta` from `Tactics.v` in the HoTT library [2].

Lemma 5.1. Let `X : Type` be a type, `F : X → Type` a type family and `P : (∀ x, F x) → F x → Type`. Suppose `a : X` is a point and `f g : (∀ x, F x)` dependent functions. Assume moreover that there exists a homotopy `H : f ∼ g` and a witness `W : P f (f a)`. Then there is a path

```
transport (λ f, P f (f a)) (path_forall f g H) W
  = transport (λ h, P h (g a)) (path_forall f g H)
      (transport (λ y, P f y) (H a) W)
```

where `path_forall f g : f ∼ g → f = g` is function extensionality.

Proof. We will replace occurrences of `H` with `apD10 (path_forall f g H)`, where `apD10` is the HoTT library name for `happly` from the HoTT book. We achieve this by transporting along the path `H = apD10 (path_forall f g H)` which comes from the propositional computation rule in section 2.9 in the HoTT book [3]. By path induction we may assume judgmental equalities `path_forall f g H` $\equiv$ $1_{\texttt{f}}$ and `f` $\equiv$ `g`, where $1_{\texttt{f}}$ `: f = f` is the identity path. It therefore suffices to show that

```
transport (λ f, P f (f a)) (path_forall f f (apD10 1_f)) W
  = transport (λ h, P h (f a)) (path_forall f f (apD10 1_f))
      (transport (λ y, P f y) (apD10 1_f a) W)
```

By definition `apD10` 1_f ≡ `(λ (x:X),` $1_{(f\ x)}$`)`, so section 2.9 in HoTT [3] provides a path

```
(path_forall f g (apD10 1_f)) = 1_f
```

and `apD10` 1_f `a` ≡ $1_{(f\ a)}$ definitionally. Using this we get

```
transport (λ f, P f (f a)) (path_forall f f (apD10 1_f)) W
  = transport (λ f, P f (f a)) 1_f W
  ≡ W
```

and

```
transport (λ h, P h (f a)) (path_forall f f (apD10 1_f))
      (transport (λ y, P f y) (apD10 1_f a) W)
  = transport (λ h, P h (f a)) 1_f
      (transport (λ y, P f y) (apD10 1_f a) W)
  ≡ transport (λ y, P f y) (apD10 1_f a) W
  ≡ transport (λ y, P f y) 1_(f a) W
  ≡ W
```

□

Part (i) of the next lemma is `transport_arrow_toconst` from `Types/Arrow.v` in the HoTT library [2]. Path (ii) is `transport_forall_constant` from `Types/Forall.v` in the HoTT library.

Lemma 5.2. Let `X Y : Type` be types. Assume there are inhabitants x_1 x_2 `: X` and a path `p :` $x_1 = x_2$.

(i) Suppose that `P : X -> Type` and `f : P` x_1 `→ Y` and `y : P` x_2. Then

```
transport (λ x, P x → Y) p f y = f (p^ # y).
```

where `p^ :` $x_2 = x_1$ denotes the inverse path and `p^ # y` ≡ `transport P p^ y` is transport along `p^`.

(ii) Let `y : Y` and `P : X → Y → Type` and `f : (∀ y, C` x_1 `y)`. There is a path

```
transport (λ x, ∀ z, P x z) p f y = transport (λ x, P x y) p (f y).
```

Proof. Both part (i) and (ii) follow from path induction. □

Part (i) of the above lemma is a version of equation (2.9.4) in the HoTT book where the codomain of `f` is non-dependent.

The proof of the next Lemma is inspired by the proof of `transport_path_universe_V_uncurried` from `Types/Universe.v` in the HoTT library [2].

Lemma 5.3. Let `X Y Z : Type` be types. If there is an equivalence `f : X ≃ Y` and function `g : X → Z` and a point `y : Y`, then

```
transport (λ (T:Type), T → Z) (path_universe f) g y = g (f⁻¹ y)
```

where f^{-1} `: Y → X` denotes the inverse function and `path_universe f : X = Y` is univalence applied to `f`.

Proof. By concatenating with the path from Lemma 5.2(i), it is sufficient to find a path of type

```
g ((path_universe f)^ # y) = g (f^-1 y)
```

where `(path_universe f)^ # y` ≡ `transport idmap (path_universe f)^ y`. It follows from section 2.10 in HoTT [3] that there is a path `f = transport idmap (path_universe f)`. Using this and path induction on `(path_universe f)`, we may assume `X ≡ Y` definitionally, and we just need to show that

```
g ((path_universe (transport idmap 1_X))^ # y) = g ((transport idmap 1_X)^-1 y)
```

Since `transport idmap 1_X` is the identity function, the right hand side above is equal to `(g y)` judgmentally. The left hand side is equal to `(g y)` propositionally because section 2.10 in the HoTT book gives a path `path_universe (transport idmap 1_X) = 1_X`, so that

```
g ((path_universe (transport idmap 1_X))^ # y) = g (1_X^ # y) ≡ g y
```

□

Given a family of equivalences `f : (∀ i, F i ≃ G i)`, function extensionality composed with univalence gives a path `F = G`. We will need the definitional equality of this:

Lemma 5.4.

```
Definition path_equiv_family {I} {F G : I → Type} (f : ∀ i, F i ≃ G i)
  : F = G
  := path_forall F G (λ i, path_universe (f i)).
```

□

5.2 Isomorphisms induce paths

In this subsection we prove the main theorem. We let `A B : Algebra` σ be two algebras (Definition 3.5) for some signature σ `: Signature` (Definition 3.1).

Lemma 5.5. Let `w : SymbolType` σ be a symbol type (Definition 3.1) Suppose α `: Operation A w` and β `: Operation B w` are operations of types given by `w`, see Definition 3.3 and the implicit coercion in definition 3.7. Let `f : (∀ (s : Sort` σ`), A s ≃ B s)` be a family of equivalences between the carrier sets of `A` and `B`. Assume `OpPreserving f` α β (Definition 4.1) holds. There is a path between the operations

```
transport (λ C, Operation C w) (path_equiv_family f) α = β.
```

where `path_equiv_family f : carriers A = carriers B` is given in Lemma 5.4.

Proof. We proceed by induction on `w`. In case `w ≡ [:t:]`, then `Operation C w ≡ C t`, for any `C : Sort` σ `→ Type`, and `OpPreserving f` α β ≡ `(f t` α `=` β`)`. Set

```
P := (λ (C : Sort σ → Type) (X : Type), X)}.
```

Then we get the following chain (see the comments below).

```
transport (λ C, Operation C w) (path_equiv_family f) α
  ≡ transport (λ C, C t) (path_equiv_family f) α
  ≡ transport (λ C, P C (C t)) (path_equiv_family f) α
  = transport (λ C, P C (B t)) (path_equiv_family f)
      (transport (λ X, P A X) (path_universe (f t)) α)
  ≡ transport (λ C, B t) (path_equiv_family f)
      (transport idmap (path_universe (f t)) α)
  = transport idmap (path_universe (f t)) α
  = (f t) α
  = β.
```

The first `=` path comes from Lemma 5.1. The second `=` path is from Lemma 2.3.5 in HoTT [3]. The third `=` path is from Section 2.10 in HoTT [3]. The last path is the assumption `OpPreserving f α β ≡ (f t α = β)`.

In case `w ≡ t ::: w'`, then `Operation C w ≡ (C t → Operation C w')`, for any `C : Sort σ → Type`, and

```
OpPreserving f α β ≡ ∀ (x : A t), OpPreserving f (α x) (β (f t x)).        (*)
```

It suffices to find a path in `Operation B w'` of type

```
transport (λ C, Operation C w) (path_equiv_family f) α y = β y
```

where `y : B t` is some inhabitant. By (∗) above with `(f⁻¹ y)` we have

```
OpPreserving f (α (f⁻¹ y)) (β y)                                          (**)
```

because

```
OpPreserving f (α (f⁻¹ y)) (β (f t (f⁻¹ y))) = OpPreserving f (α (f⁻¹ y)) (β y).
```

Write

```
P₁ := λ (C : Sort σ → Type) (X : Type), X → Operation C w'
P₂ := λ (C : Sort σ → Type) (z : B t), Operation C w'.
```

Then we have the chain

```
transport (λ C, Operation C w) (path_equiv_family f) α y
  ≡ transport (λ C, C t → Operation C w') (path_equiv_family f) α y
  ≡ transport (λ C, P₁ C (C t)) (path_equiv_family f) α y
  = transport (λ C, P₁ C (B t)) (path_equiv_family f)
      (transport (λ X, P₁ A X) (path_universe (f t)) α) y
  ≡ transport (λ C, B t → Operation C w') (path_equiv_family f)
      (transport (λ X, X → Operation A w') (path_universe (f t)) α) y
  ≡ transport (λ C, ∀ (z : B t), P₂ C z) (path_equiv_family f)
      (transport (λ X, X → Operation A w') (path_universe (f t)) α) y
  = transport (λ C, P₂ C y) (path_equiv_family f)
      (transport (λ X, X → Operation A w') (path_universe (f t)) α y)
  ≡ transport (λ C, Operation C w') (path_equiv_family f)
      (transport (λ X, X → Operation A w') (path_universe (f t)) α y)
  = transport (λ C, Operation C w') (path_equiv_family f) (α (f⁻¹ y))
  = β y.
```

The first `=` path is Lemma 5.1. The second `=` path is Lemma 5.2(ii). The third `=` path is Lemma 5.3. Using (∗∗) above, the last path follows by induction. □

Now we have the tools to prove the main theorem.

Theorem 5.6. If there is an isomorphism `f : Homomorphism A B` then `A = B`.

Proof. By Lemma 3.6 we just need to find two paths

- `p : carriers A = carriers B` in `Sort σ → Type`,
- `q : p#(operations A) = operations B` in `∀ (u : Symbol σ), Operation B (σ u)`,

where

```
p#(operations A) ≡ transport (λ C, ∀ u, Operation C (σ u)) p (operations A)
```

According to Lemma 4.5, `f : (∀ s, A s ≃ B s)` is a family of equivalences. Hence, for path `p`, we can choose `path_equiv_family f : carriers A = carriers B` from Lemma 5.4.

For path `q : p#(operations A) = operations B`, set

```
R := λ (C : Sort σ → Type) (u : Symbol σ), Operation C (σ u).
```

For any function symbol `v : Symbol σ`,

```
(p#(operations A)) v
  ≡ transport (λ C, ∀ u, R C u) p (operations A) v
  = transport (λ C, R C v) p (operations A v)
  ≡ transport (λ C, Operation C (σ v)) (path_equiv_family f) (v^^A)
  = v^^B
  ≡ operations B v.
```

The first `=` path follows from Lemma 5.2(ii). The other `=` path follows from Lemma 5.5 because `OpPreserving f (v^^A) (v^^B)` holds by Definition 4.1. □

6 Conclusions and related work

This report presented a beginning of a Universal Algebra development for the HoTT library based on Math Classes [1]. The fact that isomorphic objects are equal is one of the things that distinguish HoTT from set theoretic and category theoretic foundations. Using this main theorem we can obtain paths from the isomorphism theorems, see `https://github.com/andreaslyn/hott-classes/tree/handin/theory`.

There is a similar theorem by T. Coquand and N. A. Danielsson [4]. They work with a type `U : Type` called a universe. The universe has the role of characterising algebraic structures, similar to `Signature` in this report. This allows for more flexibility as to which algebraic structure is supported, but it requires a different definition of isomorphism. They define it by

```
Definition IsIsomorphism
  (U : Type) (El : U → Type → Type)
  (resp : ∀ {a} {A B : Type}, (A ≃ B) → El a A → El a B)
  (resp_id : ∀ {a} {A : Type} (x : El a A), resp equiv_idmap x = x)
  {a} {A B : Type} (f : A ≃ B) (x : El a A) (y : El a B)
  := resp f x = y.
```

The `U` argument is the universe. The value `El a A` is the type of the algebraic structure characterised by `a:U`. This corresponds to the type `∀ u, Operation C (σ u)` from Definition 3.3 above. The `resp f` function is for transport of structure `El a A → El a B` by an equivalence `f : A ≃ B`. The `resp_id` argument is there to make sure `resp` is well behaved. They have a transport theorem which shows that such a `resp` function together with a proof `resp_id` satisfies

```
resp f x = transport (El a) (path_universe f) x
```

An equivalence `f : A ≃ B` is an isomorphism if the algebra structure of `A` is transported by `resp f` to that of `B`. This corresponds to Lemma 5.5 in this report. They avoid using `path_forall`, as in Lemma 5.4 above, because they are working with single-sorted algebraic structures, just one carrier type. We have been working with multi-sorted algebraic structures `Carriers σ ≡ (Sort σ → Type)`, a family of carrier types.

Type theoretic developments of Universal Algebra are often using Setiods [1, 5] to cope with quotients. Later reports on this development will show that this is not needed in Universal Algebra for HoTT, so allows for a cleaner development.

My thoughts on Coq are mostly positive. When doing set theoretic mathematics I feel a need to double or triple check my arguments for correctness. This is especially true when working on abstract mathematics hard to visualise. When I have finished a proof in Coq I do not need worry about correctness of the arguments. In my opinion, this is the main advantage provided by proof assistants. The Ltac language in Coq is a great tool, although there could have been more focus on its readability. It allows to try out ideas quickly and it is most often more readable than explicit proof terms.

References

[1] B. Spitters and E. van der Weegen, "Type classes for mathematics in type theory," *ArXiv e-prints*, Feb. 2011.

[2] A. Bauer, J. Gross, P. LeFanu Lumsdaine, M. Shulman, M. Sozeau, and B. Spitters, "The HoTT Library: A formalization of homotopy type theory in Coq," *ArXiv e-prints*, Oct. 2016.

[3] The Univalent Foundations Program, "Homotopy Type Theory: Univalent Foundations of Mathematics," tech. rep., Institute for Advanced Study, 2013.

[4] T. Coquand and N. A. Danielsson, "Isomorphism is equality," vol. 24, p. 1105–1120, 11 2013.

[5] E. Gunther, A. Gadea, and M. Pagano, "Formalization of universal algebra in agda," *Electronic Notes in Theoretical Computer Science*, vol. 338, pp. 147–166, 10 2018.