# Automating Concept Equivalence in Dependent Type Theory

Floris van Doorn, Michael R. Douglas and David McAllester

AITP 2021

September 7, 2021

# Concept Equivalence

The term "cryptomorphism" was introduced by Garrett Birkoff before 1967 to refer to equivalence of concepts.

Birkoff did not give the term a formal definition.

As an example a group can be defined to consist of a set, the group operation, an identity element and an inverse operation. Or it can be defined to consist of just the set and group operation with axioms stating the existence of the identity and inverse elements.

Concept equivalence is fundamental

- to avoid redundancy in mathematical libraries;
- for semantic concept retrieval;
- for automated concept discovery.

# Cryptomorphism in Type Theory

To make the discussion concrete we can model "concepts" as types of dependent type theory.

A natural definition of cryptomorphism within type theory is that two concepts (types) are cryptomorphic if there exists a pair of lambda expressions defining a mapping and its two-sided inverse between the objects of the two classes.

In the groupoid model of type theory each type (concept) is associated with a groupoid given by the objects of the type and the isomorphisms between them.

A cryptomorphism gives an isomorphism between the associated groupoids.

# Signature-axiom class (SAC)

In Bourbaki, mathematical structures are expressed by SAC's, which consist of one or more carrier set(s), data (e.g. functions, relations, collections of subsets), and axioms.

In the proof assistant Lean we use type classes to define SAC's.

```
class Mul (G : Type u) := (mul : G → G → G)
class One (G : Type u) := (one : G)
class Inv (G : Type u) := (inv : G → G)
class Semigroup (G : Type u) extends Mul G :=
(mul_assoc : ∀ a b c : G, a * b * c = a * (b * c))
class Monoid (G : Type u) extends
  Semigroup G, One G :=
(left_mul_id : ∀ (a : G), 1 * a = a)
(right_mul_id : ∀ (a : G), a * 1 = a)
class Group (G : Type u) extends Monoid G, Inv G :=
(mul_left_inv : ∀ (a : G), a⁻¹ * a = 1)
```

# Previous Work: Theories and Views

The notion of signature-axiom class corresponds to the notion of "theory" in earlier systems such as IMPS (Farmer, Guttman and Thayer, 1993).

IMPS supports theory interpretations or "views", as a way of defining a mapping from the models of one theory to the models of another.

Müller, Kohlhase and Rabe (2018) define a similar notion of view but parameterized over the choice of meta-theory allowing mappings between theories defined in different formal systems.

# Functors

A functor is a function from instances of SAC $C$ to instances of SAC $D$ given by an explicit lambda expression. Denote a functor $F$ as $C \Rightarrow_F D$.

This lambda expression cannot use indefinite description (a.k.a. classical choice), but can use definite description (a.k.a. unique choice)

```
axiom classicalChoice : Nonempty α → α
axiom uniqueChoice : Nonempty α → Subsingleton α → α
```

# Functors

A functor is a function from instances of SAC $C$ to instances of SAC $D$ given by an explicit lambda expression. Denote a functor $F$ as $C \Rightarrow_F D$.

This lambda expression cannot use indefinite description (a.k.a. classical choice), but can use definite description (a.k.a. unique choice)

```
axiom classicalChoice : Nonempty α → α
axiom uniqueChoice : Nonempty α → Subsingleton α → α
```

A simple example is cardinality, from sets to cardinal numbers. Another important example is the "forgetful" functor which takes an instance of an SAC to its carrier set. By composing with this, the cardinality functor can be applied to general SAC's.

# Cryptomorphism

A cryptomorphism between $C$ and $D$ is a pair of functors which compose both ways to the identity. One needs to say whether this is provable equality, isomorphism or something else. We use provable equality, as in the original spirit of the definition.
We denote cryptomorphism as $C \cong_c D$.

# Cryptomorphism

A cryptomorphism between $C$ and $D$ is a pair of functors which compose both ways to the identity. One needs to say whether this is provable equality, isomorphism or something else. We use provable equality, as in the original spirit of the definition.
We denote cryptomorphism as $C \cong_c D$.

Elementary examples of cryptomorphism:

- Reordering arguments; other syntactic rewrites.
- Adding/removing redundant axioms.
- Adding/removing definable data.

```
class Group (G : Type u) extends [...], Div G :=
div_eq_mul_inv : ∀ x y : G, x / y = x * y⁻¹
[...]
```

# Example: Groups and inverses

A less elementary cryptomorphism can be obtained by leaving out data fields that are uniquely specified. For example, the definition we gave earlier of "group" specified the map from an element to its inverse. However one can prove that inverses are unique, so this map can be derived from the multiplication law.

```
class Group′ (G : Type u) extends Monoid G :=
(exists_left_inv : ∀ (a : G), ∃ (b : G), b * a = 1)
```

The same statement can be made about the identity element. So there are several cryptomorphic SAC's which define the same concept of group.

# Auxiliary structures

More generally, every mathematical concept allows defining many auxiliary structures. For example given a vector space $V$, we can define a basis. The senses in which the basis is "auxiliary" are that

- The basis is not canonically associated to $V$, and
- The axioms which a basis must satisfy in order to describe a vector space are more complicated than the original axioms.

Unless you are very careful, SACs in terms of auxiliary structure will not be cryptomorphic to the original, since it contains "more structure."

# Example: Matroids

But there are concepts for which there is more than one reasonable choice for the auxiliary structures and axioms. The prototypical example (which inspired Birkhoff to propose the idea of cryptomorphism) is the concept of a matroid (Whitney 1935).

A matroid generalizes the notion of linear independence in vector spaces. We postulate as carrier set a finite set $\mathcal{E}$, which can be thought of as elements in a vector space. We then postulate an additional structure on $\mathcal{E}$ which carries information about linear dependence:

- The independent sets $\mathcal{I} \subseteq \mathcal{P}(\mathcal{E})$.
- The bases $\mathcal{B} \subseteq \mathcal{P}(\mathcal{E})$.

Each satisfying axioms we will discuss.
There are several other additional structures we could choose.

# Axioms for matroids

The independent sets $\varnothing \neq \mathcal{I} \subseteq \mathcal{P}(\mathcal{E})$ must satisfy $A' \subset A \in \mathcal{I} \rightarrow A' \in \mathcal{I}$ and the "independent set exchange property,"

$$A, B \in \mathcal{I} \wedge |A| > |B| \rightarrow \exists x \in A \setminus B \text{ s.t. } B \cup \{x\} \in \mathcal{I}.$$

The bases $\varnothing \neq \mathcal{B} \subseteq \mathcal{P}(\mathcal{E})$ must satisfy the "basis exchange property"

$$A, B \in \mathcal{B} \wedge a \in A \setminus B \rightarrow \exists b \in B \setminus A \text{ s.t. } \{b\} \cup A \setminus \{a\} \in \mathcal{B}.$$

# Axioms for matroids

The independent sets $\varnothing \neq \mathcal{I} \subseteq \mathcal{P}(\mathcal{E})$ must satisfy $A' \subset A \in \mathcal{I} \rightarrow A' \in \mathcal{I}$ and the "independent set exchange property,"

$$A, B \in \mathcal{I} \wedge |A| > |B| \rightarrow \exists x \in A \setminus B \text{ s.t. } B \cup \{x\} \in \mathcal{I}.$$

The bases $\varnothing \neq \mathcal{B} \subseteq \mathcal{P}(\mathcal{E})$ must satisfy the "basis exchange property"

$$A, B \in \mathcal{B} \wedge a \in A \setminus B \rightarrow \exists b \in B \setminus A \text{ s.t. } \{b\} \cup A \setminus \{a\} \in \mathcal{B}.$$

The relation between $\mathcal{I}$ and $\mathcal{B}$ is

$$S \in \mathcal{B} \leftrightarrow S \in \mathcal{I} \wedge \forall x \in E, S \cup \{x\} \notin \mathcal{I}.$$

So each structure can be derived from the other, both sets of axioms are of comparable length, and it is not obvious that one is more fundamental than the other. In a real sense the matroid concept has multiple foundational definitions.

# Automated discovery of cryptomorphisms

We want an algorithm that automatically detects cryptomorphism.

Intended usage: Write a new structure and ask whether a cryptomorphic structure already exists in the library.

We currently have a prototype in Lean 4 to automatically derive basic cryptomorphisms between two structures.

`https://github.com/fpvandoorn/cryptomorphism`

# Example 1

It supports reordering and renaming fields.

```
class CommMonoid (M : Type u) extends
  Mul M, One M :=
(mul_assoc : ∀ x y z : M, (x * y) * z = x * (y * z))
(mul_comm : ∀ x y : M, x * y = y * x)
(mul_one : ∀ x : M, x * 1 = x)

class AddCommMonoid (M : Type u) extends
  Add M, Zero M :=
(add_zero : ∀ x : M, x + 0 = x)
(add_comm : ∀ x y : M, x + y = y + x)
(add_assoc : ∀ x y z : M, (x + y) + z = x + (y + z))

#eval cryptomorphic CommMonoid AddCommMonoid
-- CommMonoid and AddCommMonoid are cryptomorphic
```

# Example 2

It supports packing and unpacking data fields and axioms.

```
class CommMonoid (M : Type u) extends
  Mul M, One M :=
(mul_assoc : ∀ x y z : M, (x * y) * z = x * (y * z))
(mul_comm : ∀ x y : M, x * y = y * x)
(mul_one : ∀ x : M, x * 1 = x)

class BundledCommMonoid (M : Type u) :=
(data : (M → M → M) × M)
(mul_axioms : (∀ x y, data.1 x y = data.1 y x) ∧
  (∀ x y z, data.1 (data.1 x y) z =
    data.1 x (data.1 y z)))
(mul_one : ∀ x : M, data.1 x data.2 = x)

#eval cryptomorphic CommMonoid BundledCommMonoid
-- CommMonoid and BundledCommMonoid are cryptomorphic
```

# Example 3

It supports axioms that can be (easily) derived from each other.

```
class CommMonoid (M : Type u) extends
  Mul M, One M :=
(mul_assoc : ∀ x y z : M, (x * y) * z = x * (y * z))
(mul_comm : ∀ x y : M, x * y = y * x)
(mul_one : ∀ x : M, x * 1 = x)

class CommMonoid′ (M : Type u) extends
  Mul M, One M :=
(mul_assoc : ∀ x y z : M, (x * y) * z = x * (y * z))
(mul_comm : ∀ x y : M, x * y = y * x)
(one_mul : ∀ x : M, 1 * x = x)

#eval cryptomorphic CommMonoid CommMonoid′
-- CommMonoid and CommMonoid′ are cryptomorphic
```

# Algorithm

(1) Split data fields into components;
(2) Pair up data fields that have matching type;
    - In case of ambiguity: choose fields with the same name;
(3) Try to prove each axiom of one structure from the axioms of the other structure

# Future Work

Increase the number of cryptomorphisms we recognize.

- Recognize definable data fields
- Recognize implicitly specified data fields (e.g. postulating the existence of inverses).

We want to efficiently find a structure in a large database that is cryptomorphic to a given structure.
For this it is helpful to partially "normalize" structures.

Thank you!