Normalization functions for quotient types in Coq

(Funkcje normalizujące dla typów ilorazowych w Coqu)

Marek Bauer

Praca magisterska

Promotor: dr Małgorzata Biernacka

Uniwersytet Wrocławski
Wydział Matematyki i Informatyki
Instytut Informatyki

24 października 2023

Abstract

Despite many applications for quotient types, Coq does not have built-in support for them. This thesis will discuss how to mitigate this problem by defining quotient-like types in which precisely one element exists for each abstraction class. We will focus on two approaches: the first relies on subtyping, while the second involves defining inductive types based on normalization traces. Additionally, we will mention other methods of defining quotient types, such as using setoids, additional axioms, or higher inductive types.

Pomimo wielu zastosowań dla typów ilorazowych, Coq nie posiada wbudowanego wsparcia dla nich. W tej pracy omówimy jaki sposób można zniwelować ten problem poprzez definiowanie typów prawie ilorazowych – w których dla każdej klasy abstrakcji istnieje dokładnie jeden element. Skupimy się na dwóch podejściach: pierwszy z nich polega na zastosowaniu pod-typowania, drugi natomiast na definiowaniu typów induktywnych opartych o ślady normalizacji. Ponad to zostaną wspomniane inne sposoby na definiowanie typów ilorazowych takie jak użycie setoidów, dodatkowych aksjomatów lub wyższych typów induktywnych.

Contents

Chapter 1

Introduction

1.1 The purpose

The purpose of this master thesis is to analyze methods for defining quotient types in Coq since there is no built-in method for implementing such constructions in this programming language. We seek ways to define quotient-like types similar to quotient types, where elements are considered equal if and only if they are in a quotient-defining relation. We will focus on quotient types with a normalization function and provide examples without a computable normalization function. For this purpose, we will use methods such as subtyping, inductive types based on traces of normalization functions, and more.

1.2 Prerequisites

This thesis assumes basic knowledge of algebraic concepts such as modular groups, fractions, and the Euclidean algorithm. More advanced constructions will be defined, like dividing sets by equivalence relations. The main focus is the Coq language, so basic knowledge of this programming language is necessary. The reader should be able to write definitions in Coq and differentiate between `Fixpoint` and `Definition`. Understanding how the termination checker works or knowing the sort hierarchy can aid in understanding this thesis fully, but it is optional. The reader is also expected to know the basic concepts of Type Theory, such as the product and sum of types. Additionally, optional sections include category theory and homotopy-type theory concepts. Any advanced construction from those fields needed to comprehend these sections will be defined in this thesis.

1.3 Coq as a proof assistant

Coq is a formal proof management system released under the GNU LGPL license. It was first introduced in 1984 and was based on the Calculus of Constructions, which relied on Type Theory. In 1991, Coq began using the more advanced Calculus of Inductive Constructions [27] [43], which supports higher-order logic, dependent types, and statically typed programming, among other features, all thanks to the Curry-Howard isomorphism [26].

Coq can perform simple computations. However, it was primarily developed for program extraction. Users can use functions proven in Coq in other programming languages, such as OCaml. In the Coq type system, every term has its type, and every type has its sort. This is an essential concept, as different sorts serve different purposes. There are four important sorts:

`Prop` – intends to be the sort of logical prepositions. This sort is impredicative, which means propositions can quantify over all propositions. Imprecativity allows us to write propositions that say something about all propositions. Elements of this sort are removed during code extraction. Consequently, there are limitations on eliminating proofs of prepositions during type construction.

`SProp` – similar to `Prop`, this sort is also intended to contain logical prepositions. The main difference is the definitional irrelevance of propositions. Irrelevance means that proofs of the same preposition in this sort are, by definition, equal.

`Set` – intends to be a sort for small types, like boolean values or natural numbers, but also products, subsets, and function types over these data types. Elements of this sort are expected to be preserved during code extraction. Accordingly, computations should take place in this sort.

`Type` – contains small and large sets like `Prop` and `Set`. Therefore, every type and sort lives in `Type`, for example, `Prop : Type`.

Advanced Coq C-1.3.1: Hierarchy of sorts

More experienced readers might notice that this definition of `Type` seems wrong. We know only sorts at the bottom of the hierarchy can contain itself [18]. `Type` is not `Prop` but it contains `Prop`, therefore it not at the bottom of hierarchy. Despite this, when Coq is asked what is the type of `Type`, it answers `Type : Type`. In reality, there is a whole infinite hierarchy of sorts in Coq, and every `Type` has its level, which is just hidden from users for convenience. Therefore, `Type : Type` should be read as `Type(n) : Type(n+1)`.

Coq allows users to define only functions that terminate. For many simple functions, the termination checker automatically deduces that the function terminates.

However, sometimes it requires much work to convince it. The other consequence of this is the fact that in Coq, reasoning about partial functions is difficult. Nevertheless, later we will learn how to represent non-terminating computations. The most significant advantage of Coq is that it has an imperative tactic language, which makes proving theorems much more accessible than in languages like Agda [35] or Idris [42].

1.4 Quotient types

Quotient type is a concept from abstract algebra [22], which has found applications in other fields like computer science.

Definition D-1.4.1: Quotients in abstract algebra

In abstract algebra, an underlying set T partitioned by equivalence relation $(\sim)$ is called a *quotient* and denoted as $T/\sim$.

Example E-1.4.1: Clock operations

An excellent example of a quotient is an analog clock, more precisely, the sequence of consecutive hours on it. The operations on the clock can be modeled by operations in additive integer group modulo 12 (we denote it as $\mathbb{Z}_{12}$). Almost everyone has experience with time tracking, and it is not surprising that after 12 o'clock, there is 1 o'clock. This phenomenon is caused by the fact that numbers 1 and 13 both give the same remainder when divided by 12 and that is why they are equivalent in the context of time tracking.

$$\ldots \equiv -11 \equiv 1 \equiv 13 \equiv 25 \equiv 37 \equiv \ldots \pmod{12}$$

As expected, 1:00 and 13:00 are the same on an analog clock.

1.4.1 The equivalence relation

In order to formalize quotient types, first, we need to define equivalence relations and check their properties.

Definition D-1.4.2: Equivalence relation

```
Class equivalance_relation {A : Type} (R : A -> A -> Prop) := {
  equiv_refl  : forall x : A, R x x;
  equiv_sym   : forall x y : A, R x y -> R y x;
  equiv_trans : forall x y z : A, R x y -> R y z -> R x z;
}.
```

Advanced Coq C-1.4.1: Class

In Coq `Class` is a keyword defining typeclasses known, for example, from Haskell [31]. They are used for defining interfaces for types. For someone unfamiliar with the concept of typeclasses, it can be considered as a classical Coq `Definition` with a slightly more user-friendly application.

Equivalence relation (defined in D-1.4.2) is a generalization of the most fundamental relation: equality. Similarly, we want every element to be equivalent to itself. This property is called *reflexivity*. Moreover, we want this relation to be *symmetric*, which means that $(a \sim b)$ is equivalent to $(b \sim a)$. The last property is *transitivity*. For symmetric relations, it says that when two elements, a and b, are both equivalent to a third one, then a is equivalent to b. Relations with those three properties induce equivalence classes [22] over the underlying set.

Definition D-1.4.3: Equivalence class

For an underlying set T and an equivalence relation $(\sim)$, the set of all elements equivalent to $a \in T$ is called *equivalence class* and denoted as $[a]_\sim$.

$$[a]_\sim = \{x \in T : x \sim a\}$$

A quotient $T/\sim$ is a set of all equivalence classes.

$$(T/\sim) = \{[t]_\sim : t \in T\}$$

Example E-1.4.2: Fraction's notations

An example of a set with an equivalence relation is the set of fraction's notations. We know that every fraction can be represented in infinitely many ways. Therefore, we can define the equivalence relation in which all representations of the same fraction.

$$\frac{1}{2} = \frac{2}{4} = \frac{3}{6} = \frac{4}{8} = \frac{5}{10} = \ldots$$

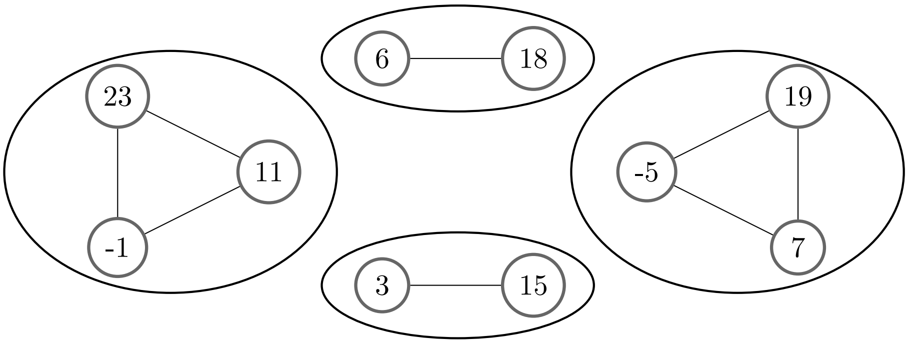
Example E-1.4.3: Modular arithmetic

Another example is a modular group from Example E-1.4.1. We showed that multiple integers have the same value in modular arithmetic. Hence, all numbers with the same value are in the same equivalence class.

$$\cdots \equiv -11 \equiv 1 \equiv 13 \equiv 25 \equiv 37 \equiv \ldots \pmod{12}$$

Visualization V-1.4.1: Equivalence classes

Visualization of equivalence classes for selected elements of group $\mathbb{Z}_{12}$. Lines represent elements in equivalence relation, and ellipses represent equivalence classes:



1.4.2 Type-theoretic view

This thesis focuses on the implementation of quotient types in Coq. Thus, we focus on this concept from the type theory perspective. In type theory, quotient types are also induced by some equivalence relation ($\sim$) on underlying type T and denoted as $T/\sim$. We denote the equality of elements in the quotient type $T/\sim$ as $a =_\sim b$. Two elements a and b are equal ($a =_\sim b$) if and only if they are in the relation ($a \sim b$). Every element of the underlying type T is also an element of quotient type $T/\sim$.

Nevertheless, only some functions from underlying type T are well-defined from type $T/\sim$. Function $f : T \to X$ defined on the underlying types is also a well-defined function $f : (T/\sim) \to X$ on the quotient type, if $a \sim b$ implies that $f(a) = f(b)$. Without this restriction, we could differentiate between elements of the same equivalence class by applying an ill-defined function. In other words, applying an ill-defined function breaks the rule that says: $x = y \Rightarrow f(x) = f(y)$.

1.5 Quotient types in Coq

As we mentioned previously, Coq has no built-in method for quotient constructions. Nevertheless, quotients are a helpful concept in formalizing mathematics. Moreover, quotient data types such as sets and multisets are used in many algorithms. Therefore, since Coq was introduced, users have been looking for ways to deal with quotients in the Coq language [25] [17] [19]. From those studies, two ways of working with quotient-like types emerged. The first one focuses on replacing the equality relation with an equivalence relation. The second focuses on changing the underlying type so that equivalence implies equality. Both approaches have their disadvantages and both are used depending on the situation. In this paper, we focus on the second approach.

1.5.1 The setoid approach

Definition D-1.5.1: Setoid

In mathematics, a *setoid* is a set A with an equivalence relation $(\sim)$. It is denoted as $(A, \sim)$. In setoids, relation $(\sim)$ is meant to be used in place of equality on set A.

Setoids are concepts known from type theory [7] [3]. In Coq, they are often used in the formalization of mathematical concepts. When working with setoids, we need to replace the equality relation in the theorem we are proving with the equivalence relation of the setoid we want to use. It is a minor inconvenience. The bigger one occurs when we want to apply a function on an element of our setoid. In this case, we must prove that the applied function is well-defined on setoid (respects equivalence relation). The Coq standard library has tools to help users use setoids, like rewriting parts of equivalent statements. Unfortunately, it is still far more problematic than using simple equality.

1.5.2 The quotient-like type approach

Another approach to the lack of quotients in Coq is to define quotient-like types where equivalence is the same as equality. This approach usually involves reducing each equivalence class to a single element.

Subtyping

This concept will be discussed later in Chapter 2, so the formal definition will be skipped here. Its main idea is to construct a type containing a specific subset of elements. Those elements need to satisfy a particular predicate in the case of the quotient – the predicate of being normalized.

A tailor-made inductive type

This approach usually gives the best results but only applies to some quotient types. Chapter 4 will show examples of such types based on traces of normalization function. Unfortunately, no general way of defining such type out of the normalization process has been found.

Additional axioms

To the Coq system, we can add axioms, and by doing so, we can define quotients similarly as in other languages like Lean [4]. This particular construction will be shown in Chapter 6. Unfortunately, by adding axioms, we lose the computability of our proofs. However, usually, we do not need computable proofs. Moreover, it's essential to exercise caution to avoid introducing contradictions into the system.

Example E-1.5.1

When dealing with arithmetic modulo 2, we might be tempted to add the following axiom:

```
Axiom Modulo2 : forall n: nat, n = S (S n).
```

It lets us effortlessly rewrite numbers back to their normalized form. Nevertheless, by mistake, we introduced a contradiction. Let us define a function `isZero`, which returns `true` for zero and only zero. Therefore `true = isZero(0) = isZero(2) = false`, since `0 = 2`.

Private inductive types

Another way to construct quotients is by using private inductive types [11]. By using them, we can define the type on which pattern matching outside the module where it is defined is forbidden. If it is done correctly, it enables us to create constructions like in Example E-1.5.1 without introducing contradiction since a user cannot define ill-defined functions.

1.6 The equivalence relation induced by a function

Theorem T-1.6.1: The equivalence relation induced by a function

Every function $h : T \to B$ induces a specific equivalence relation $(\sim_h)$ defined below:

$$\forall x, y \in T, x \sim_h y \iff h(x) = h(y).$$

Moreover, every function $g : B \to X$ induces a well-defined function $f : T/\sim_h \to X$ defined as $f = g \circ h$.

Proof P-1.6.1: The equivalence relation induced by a function

Proof that $(\sim_h)$ is an equivalence relation follows directly from the fact that equality on type B is an equivalence relation. □

Given $a, b \in T$ such that $a \sim_h b$. By definition of $(\sim_h)$ we know that $h(a) = h(b)$ therefore $f(a) = g(h(a)) = g(h(b)) = f(b)$. So f respects relation $\sim_h$. □

1.6.1 Definition of a normalization function

This paper focuses on the particular case of the function $h : T \to T$, which is idempotent. Such functions we call normalization functions.

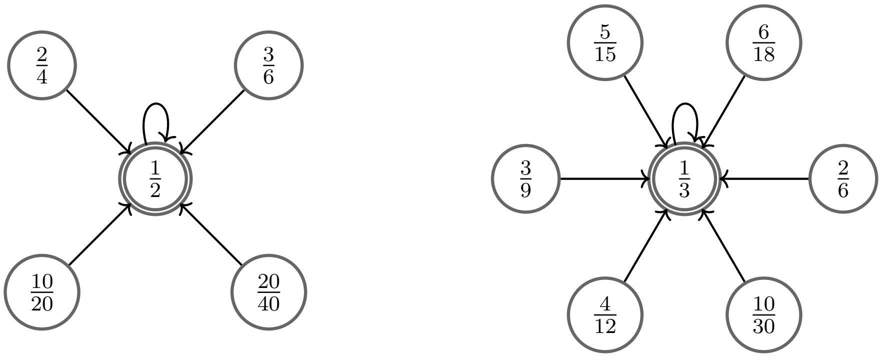
Definition D-1.6.1: Normalization function

```
Class normalization {A : Type} (f : A -> A) :=
  idempotent : forall x : A, f (f x) = f x.
```

As we know, equivalence relations partition elements into equivalence classes. A normalization function selects a single element from each equivalence class. Elements that have been selected are referred to as normalized or elements in normal form. The idempotence restriction is required to ensure that the normalization function does not change elements that are already in normal form.

Visualization V-1.6.1: Normalization function

Example of applying normalization function to rational numbers represented as $\mathbb{Z} \times \mathbb{N}_+$:



1.6.2 Examples of normalization functions

Example E-1.6.1: Rational numbers

The standard representation of rational numbers is $\mathbb{Z} \times \mathbb{N}$. For such representation, we can define elements in normal form as irreducible fractions. Therefore, the normalization function should reduce the fraction by dividing the numerator and the denominator by their greatest common divisor.

Example E-1.6.2: Unordered pair

We can define a normalization function for unordered pairs using linear order defined for the underlying type, for example, pairs of natural numbers. Such a function uses the linear order to sort elements of a pair. By doing so, the original order of elements in the pair is lost.

Example E-1.6.3: Integers modulo n

The normalization function for numbers in arithmetic modulo n is as we expect the operation of calculating modulo.

1.7 Quotient types without a normalization function

Normalization functions are an excellent tool for defining quotient-like types. By using them, we can select a representative for each equivalence class. Unfortunately, not every quotient type has a computable normalization function. With the axiom of choice, we can always select a set of representatives of each equivalence relation. Therefore, we can always define the normalization function. However, in this thesis, we try to avoid utilizing additional axioms, since they make functions uncomputable.

1.7.1 Unordered pairs

An unordered pair is one of the most basic quotients for which a normalization function does not exist. A normalization function exists if an underlying type has a decidable linear order. However, in the general case, it is impossible to define the normalization function. Moreover, for some types like, for example, $\mathbb{N} \to \mathbb{N}$, it is also possible to create such a normalization function based on the computable minimum and maximum, although linear order is not decidable [2].

Theorem T-1.7.1

There is no function $f : (A \times A) \to A \times A$ for any type A that, for any $\bigcirc, \square \in A$ we have $f((\bigcirc, \square)) = f((\square, \bigcirc))$ and $\bigcirc, \square \in f((\bigcirc, \square))$

Proof P-1.7.1

Since f is a function for every type A, we cannot use values of $\square, \bigcirc$ to determine the order of elements. Therefore, there are only two functions: $f((\bigcirc, \square)) = (\bigcirc, \square)$ and $f((\bigcirc, \square)) = (\square, \bigcirc)$, that satisfy the second property. As can be easily checked, neither of them satisfies the first property. $\square$

1.7.2 Real numbers represented by Cauchy's sequences

Definition D-1.7.1: Cauchy sequance

Sequence $\{a_n\}_{n \in \mathbb{N}}$ in called *Cauchy sequance* if elements become arbitrarily close to each other as the sequence progresses [14].

$$(\forall \epsilon > 0).\ (\exists N \in \mathbb{N}^+).\ (\forall m, n > N).\ |a_n - a_m| < \epsilon$$

Cauchy sequences are often used to construct real numbers out of rational numbers [37]. The issue with this construction is that infinitely (even uncountably) many sequences represent a single real number. Using quotients to fix this problem would give us an excellent representation of real numbers. Unfortunately, there is no computable normalization function for Cauchy sequences.

Theorem T-1.7.2

There is no computable function $f : (\mathbb{N} \to \mathbb{Q}) \to (\mathbb{N} \to \mathbb{Q})$ that, for any $a, b \in (\mathbb{N} \to \mathbb{Q})$ we have $\lim_{n \to \infty} a_n = \lim_{n \to \infty} b_n \iff f(a) = f(b)$.

Proof P-1.7.2

Let's assume such function f exists. Let s_k be a sequence consisting of ones to k-th place and zeros after. Let s_∞ be a sequence consisting of all ones.
For all k limit of s_k is zero, and limit of s_∞ is one. Therefore $f(s_k) \neq f(s_\infty)$. If they are different, that means there is a number $t \in \mathbb{N}$ for which $f(s_k)_t \neq f(s_\infty)_t$. So, we are able to differentiate between those two in finite time.
Let M be the initial state of the universal Turing machine. Let $c_M(t)$ be zero if the Turing machine halts before t-th step and one otherwise. This function is computable for every M, so we can apply a function f to it and check if the Turing machine with initial state M halts. But the halting problem is undecidable [20], so such function f cannot exist. ↯

1.7.3 The delay monad

Definition D-1.7.2: Delay monad

```
CoInductive delay (A : Type) : Type :=
| output : A -> delay A
| wait   : delay A -> delay A.
```

The delay monad [1] represents computations that may or may not terminate. It is a valuable concept in Coq since it provides an entirely safe way of implementing functions that may not terminate. We want our normalization function to return a value immediately if the computation ever terminates and return infinite delay otherwise. Unfortunately, similar to Cauchy sequences, such a normalization function does not exist.

Theorem T-1.7.3

There is no computable function $f : \text{delay } A \to (A + \text{unit})$ that, for any terminating computation, returns its final value. Otherwise, it returns an element of unit type.

Proof P-1.7.3

Let's assume that such function f exists. We can model the computation of every partial computable function using a delay monad. Using the normalization function f, we can determine if a partial function terminates. This makes the halting problem decidable, but we know it is undecidable [20]. Therefore, such a normalization function cannot exist. ↯

Chapter 2

Quotient types and subtypes

This chapter describes how subtyping can be used to define quotient-like types. It focuses on issues that may appear while using this approach in Coq and shows how to deal with them.

2.1 Subtyping

Definition D-2.1.1: Subtypes

A *subtype* $A \sqsubseteq B$ is a type that contains a specific subset of elements of the underlying type B.

Example E-2.1.1

A great example of a subtype is the type of even numbers, which is a subtype of natural numbers.

Subtyping is a pretty intuitive concept. However, subtyping in Coq can confuse users accustomed to object-oriented languages like Java [15]. In such languages, subtyping is usually associated with the concept of inheritance and is used to define polymorphic construction. In other words, we can use an element of a subtype when an element of the underlying type is required. Similarly, in mathematics, we expect an even number to be a natural number. Unfortunately, such intuitions do not apply in Coq.

Definition D-2.1.2: Subtypes in Coq

```
Inductive sig (A : Type) (P : A -> Prop) : Type :=
    exist : forall x : A, P x -> sig P.
```

Definition D-2.1.3: Subtypes in Coq using record

```
Record sig' (A : Type) (P : A -> Prop) : Type := exist' {
    proj1' : A;
    proj2' : P proj1';
}
```

Subtypes in Coq can be defined in two equivalent ways (see definitions D-2.1.2 and D-2.1.3). The first definition is used in Coq's standard library. The second one nicely shows that a subtype is actually a dependent pair built of two elements – a value and the proof that the value satisfies a subtype-defining predicate.

Definition D-2.1.4: Dependent pair

Dependent pair is a generalization of a pair where the type of the second element depends on the value of the first one.

Example E-2.1.2

An example of a dependent pair type is the type of pairs of two coprime numbers. The value of the first element determines the type of the second element. For example, if the first element is five, the second element must be a number coprime to five.

Since subtypes are defined as dependent pairs in Coq, we cannot use an element of the subtype as an element of the underlying type. However, in contrast to the majority of programming languages, in Coq we can use subtypes to define concepts like the type of sorted lists and, for example, use this type as a type of argument of the binary search function. The benefits of such function types are obvious for developing reliable software.

Advanced Coq C-2.1.1: The notation for subtyping

In Coq, there is a special notation for subtypes: `{a : A | P a}` that should resemble mathematical notation of $\{a \in A : P(a)\}$.

Example E-2.1.3: The subtype of natural numbers less than 10

```
Definition lessThanTen := {a : nat | a < 10}.
```

2.1.1 Connection to quotient types

Subtyping is the dual construction of quotient types. As the introduction mentions, there is no built-in method for implementing quotient types in Coq. Therefore, we will use subtypes to define quotient-like types containing only normalized elements of the underlying type. The normalized elements are defined as the image of some normalization function. This construction defines a type where equality is equivalent to a quotient-defining equality relation because every equivalence class is reduced to a single element.

Definition D-2.1.5: Subtype of normalized elements

```
Record quotient {A : Type} (f : A -> A) `{normalization f} := {
  val          : A;
  valIsNormal : val = f val;
}.
```

Advanced Coq C-2.1.2: The backtick notation

In Coq, we do not need to name all assumptions, if they are not needed explicitly in the definition. In this case, we can use the backtick notation `` `{property} ``.

2.2 Duality

In the previous section, we mentioned that subtyping is a dual concept to quotient types. This optional section explains what it means and why they are considered dual. Knowledge about category theory would improve the reading experience. However, this part is more of an interesting fact than an integral part of this work, so feel free to skip it.

Definition D-2.2.1: Duality

In category theory, the statement *dual* to σ is denoted as σ^{op} and defined by [32]:

- interchanging the source and target of each morphism in σ,
- interchanging the order of composing morphisms in σ.

Visualization V-2.2.1: Product and coproduct

The formal definition of duality might take much effort to understand. A visual example of two dual constructions should give enough insight to comprehend this concept. On the left side, we see a product, and on the right side, a dual construction to product, known as a coproduct.

$$\begin{tikzcd}
& X \arrow[dl, "f_1"'] \arrow[d, dashed, "f"] \arrow[dr, "f_2"] & \\
A & A \times B \arrow[l, "\pi_1"] \arrow[r, "\pi_2"'] & B
\end{tikzcd}
\qquad
\begin{tikzcd}
& X & \\
A \arrow[ur, "f_1"] \arrow[r, "\eta_1"'] & A \oplus B \arrow[u, dashed, "f"'] & B \arrow[ul, "f_2"'] \arrow[l, "\eta_2"]
\end{tikzcd}$$

Both diagrams commute, meaning that all directed paths in the diagram with the same start and endpoints lead to the same result [32].

In other words, we can construct a dual concept by reversing each arrow (morphism) in the diagram. However, definitions of subtyping and quotient types do not seem to have any arrows. We must look into pushout and pullback constructions known from category theory [32] to find them.

2.2.1 Pushout

Definition D-2.2.2: Pushout

The *pushout* [32] P is defined by two morphisms $f : X \to A$ and $g : X \to B$, with common domain X.

$$\begin{tikzcd}
X \arrow[r, "g"] \arrow[d, "f"'] & B \arrow[d, "\eta_2"] \arrow[ddr, bend left, "\zeta_2"] & \\
A \arrow[r, "\eta_1"'] \arrow[drr, bend right, "\zeta_1"'] & P \arrow[dr, dashed, "u"] & \\
& & Q
\end{tikzcd}$$

The pushout P is an object along with two morphisms $\eta_1 : A \to P$ and $\eta_2 : B \to P$ that complete commutative square with two given morphisms $f : X \to A$ and $g : X \to B$. However, not every object that makes the square commutative is a pushout. The pushout must be the "most general" way to complete the commutative square. By the "most general" way, we mean that for every other object Q, which also completes commutative square with two morphisms $\zeta_1 : A \to Q$ and $\zeta_2 : B \to Q$, a unique morphism $u : P \to Q$ must exist for which the diagram commute. Not for all morphisms $f : X \to A$ and $g : X \to B$, pushout exists, but if it exists, it is unique up to the unique isomorphism.

The pushout is a useful concept in category theory, but its connection to quotient types is not obvious. To see this connection let us take a trip to the world of *Set* category. In this category, sets are objects, and functions between sets are morphisms. We will show that, indeed, in this category, quotient types are pushouts. On the diagram of the pushout definition, we can see that A, B, and P define something in the shape of a coproduct. Let us use coproduct $A \oplus B$ as a base for the pushout definition and let functions $\eta_1 : A \to A \oplus B$ and $\eta_2 : B \to A \oplus B$ be natural injections. For $A \oplus B$ to be pushout, we also need to ensure that X, A, B and $A \oplus B$ create a commutative square, that means for every $x \in X$ following should be true $\eta_1(f(x)) = \eta_2(g(x))$. In order to achieve that, we need to modify $A \oplus B$, by gluing together injections of $f(x)$ and $g(x)$ in $A \oplus B$. As we know, gluing together elements means defining the quotient type $P = ((A \oplus B)/ \sim)$, where for every $x \in X$ $f(x) \sim g(x)$. For $P = ((A \oplus B)/ \sim)$ to be a pushout, we need to define $(\sim)$ as the largest in terms of the number of equivalence classes relation that completes the commutative square.

Theorem T-2.2.1

If there exists a larger equivalence relation $(\simeq)$ for which X, A, B, and $Q = ((A \oplus B)/ \simeq)$ also commutes, then there cannot exist a unique morphism $u : P \to Q$ that makes the whole diagram commutative.

Proof P-2.2.1

Since $|Q| > |P|$, morphism $u : P \to Q$ cannot be a surjection. Therefore, there exists $y \in Q$ such that $(\forall x \in P)\ u(x) \neq y$. Without loss of generality exists $a \in A$ such that $\zeta_1(a) = y$. From commutivity we know that $\zeta_1(a) = u(\eta_1(a))$, but $\eta_1(a) \in P$. $\lightning$

Example E-2.2.1: Circle defined by a pushout

In category *Set*, we can use a pushout to construct a circle out of two bounded lines $[0, 1]$. Let C be a pushout defined by two identity morphisms $\mathrm{id} : \{0, 1\} \to [0, 1]$.

$$\begin{tikzcd}
\{0,1\} \arrow[r, "\mathrm{id}"] \arrow[d, "\mathrm{id}"'] & {[0,1]} \arrow[d, "\eta_2"] \\
{[0,1]} \arrow[r, "\eta_1"'] & C
\end{tikzcd}$$

Let us check if C is a circle. Since the diagram above commutes, then we know that $\eta_1(0) = \eta_2(0)$ and $\eta_1(1) = \eta_2(1)$ so, we glued together two points. Moreover, we know that C is defined by the largest equivalence relation in terms of equivalence classes. Therefore, every other point of both bounded lines has to have a natural injection into C. Based on this, we can conclude that C represents a circle. This construction can be generalized for n-dimensional hyper-spheres. Gluing together two n-dimensional hemispheres alongside $(n-1)$-dimensional sphere, gives a n-dimmensional sphere.

2.2.2 Pullback

Knowing that a pullback is a dual construction to pushout and knowing what dual construction is, everyone should be able to draw a diagram that defines pushout.

Definition D-2.2.3: Pullback

The pullback [32] P is defined by two morphisms $f : A \to X$ and $g : B \to X$, with common codomain X.

$$\begin{tikzcd}
Q \arrow[dr, dashed, "u"] \arrow[drr, bend left, "\zeta_2"] \arrow[ddr, bend right, "\zeta_1"'] & & \\
& P \arrow[r, "\pi_2"] \arrow[d, "\pi_1"'] & B \arrow[d, "g"] \\
& A \arrow[r, "f"'] & X
\end{tikzcd}$$

The pullback P is an object along with two morphisms $\pi_1 : P \to A$ and $\pi_2 : P \to B$ that complete commutative square with two given morphisms $f : A \to X$ and $g : B \to X$. However, not every object that makes the square commutative is a pullback. The pullback must be the "most general" way to complete the commutative square. By the "most general" way, we mean that for every other object Q, which also completes commutative square with two morphisms ζ_1 :

$Q \to A$ and $\zeta_2 : Q \to B$, a unique morphism $u : Q \to P$ must exist for which the diagram commute. Not for all morphisms $f : A \to X$ and $g : B \to X$, pullback exists, but if it exists, it is unique up to the unique isomorphism.

To understand the connection between a pullback and subtyping, we take a trip to the *Set* category again. As expected in dual construction, in the pullback A, B, and P are in the shape of a product. Therefore, let us use product $A \times B$ as the base for the pushout definition and let functions $\pi_1 : A \times B \to A$ and $\pi_2 : A \times B \to B$ be natural projections. In order to complete the commutative square for every $p \in A \times B$, the following needs to be true $f(\pi_1(p)) = g(\pi_2(p))$. Therefore, we need to use the following subset $P = \{(a,b) \in A \times B : f(a) = g(b)\}$.

Theorem T-2.2.2

For every other subset $Q \subseteq A \times B$ and two morphisms $\zeta_1 : Q \to A$ and $\zeta_1 : Q \to B$ that complete the commutative square, there exists a unique morphism $u : Q \to P$, which makes the whole diagram commutative.

Proof P-2.2.2

Let define $u(q) = (\zeta_1(q), \zeta_2(q))$. Since π functions are simple projections the following is true $\zeta_1 = \pi_1 \circ u$ and $\zeta_2 = \pi_2 \circ u$. □

Example E-2.2.2: Pairs of integers with the same parity

In category *Set*, we can use the pullback to define the set of pairs of integers with the same parity. Let us define morphism $f(n) = n \bmod 2$ that checks the parity of an integer.

$$\begin{tikzcd}
P \arrow[r, "\pi_2"] \arrow[d, "\pi_1"'] & \mathbb{Z} \arrow[d, "f"] \\
\mathbb{Z} \arrow[r, "f"'] & \{0,1\}
\end{tikzcd}$$

As we know, the "most general" P that completes this commutative square is $\{(n,m) \in \mathbb{Z} \times \mathbb{Z} : n \equiv_2 m\}$.

2.2.3 Conclusion

As shown in the examples above, the pushout can define quotient types in category theory, and the pullback can be used to define subtypes. Those two constructions are mutually dual. Therefore, we can call subtyping a dual concept to quotient types.

2.3 Uniqueness of representations

In Coq, subtyping is accomplished by using dependent pairs of value and proof that the value satisfies the subtype-defining predicate. Unfortunately, this means that only some properties of subtypes known from mathematics hold in Coq. As it was mentioned at the beginning of this chapter, we cannot use a member of a subtype as a member of the underlying type. This particular problem is easy to fix using projections. The other problem of Coq's subtypes related to using subtyping as quotient-like types is that we cannot prove that only one member of the subtype exists for every value that satisfies the normalization predicate. As we know, the irrelevance of proofs is independent of the axioms of the `Prop` sort [12]. It is a major problem since we want to use subtyping to reduce each equivalence class to a single element. The possibility of the existence of multiple elements that only differ by proof of subtype defining predicate makes this construction useless for quotient types applications. Hence, we need to define the subset of quotient types for which its subtype representations are unique.

Definition D-2.3.1: Class of subtypes with unique representations

```
Class unique_representations {A : Type} (P : A -> Prop) :=
  uniq : forall (x y : {a : A | P a}), proj1_sig x = proj1_sig y
           -> x = y.
```

Advanced Coq C-2.3.1

In Coq function `proj1_sig` is a projection of the subtype value. It can be thought of as a lifting to the underlying type.

2.3.1 Axiomatic approach

As mentioned in the introduction, we avoid using axioms in this paper. However, it is worth considering their effect on the uniqueness of representations problem and whether using them is safe.

Proof irrelevance axiom

Proof irrelevance is a known concept in computer science [6]. It states that every two proofs of the same statement are equal.

Definition D-2.3.2: Axiom of proof irrelevance

```
Definition Irrelevance := forall (P : Prop) (x y : P), x = y.
```

Using this axiom, we can easily prove that every subtype has a unique representations (as defined in D-2.3.1).

Theorem T-2.3.1

```
Theorem irrelevance_impl_unique_representation : Irrelevance ->
  forall (A : Type) (P : A -> Prop), unique_representations P.
```

Proof P-2.3.1

```
intros Irr A P. constructor. intros [x xp] [y yp] H.
cbn in H. subst.
Require Import Coq.Logic.EqdepFacts.
apply eq_dep_eq_sig.
specialize (Irr (P y) xp yp); subst.
constructor. Qed.
```

The uniquness of represenations (defined in D-2.3.1) every subtype is equivalent to the proof irrelewance axiom. Thus, the axiom of proof irrelevance is required for every subtype to have a unique representations.

Theorem T-2.3.2

```
Theorem unique_representation_impl_irrelevance :
  (forall (A : Type) (P : A -> Prop), unique_representations P)
    -> Irrelevance.
```

Proof P-2.3.2

```
intros Uniq P x y.
specialize (Uniq unit (fun _ => P)). destruct Uniq as [Uniq].
specialize (Uniq (exist _ tt x) (exist _ tt y) eq_refl).
Require Import Coq.Logic.Eqdep_dec.
refine (eq_dep_eq_dec (A := unit) _ _).
- intros. left. destruct x0, y0. reflexivity.
- apply eq_sig_eq_dep. apply Uniq. Qed.
```

Advanced Coq C-2.3.2: Dependent equality

In coq, *dependent equality* is defined as:

```
Inductive eq_dep (U : Type) (P : U -> Type) (p : U) (x : P p)
  : forall q : U, P q -> Prop :=
    eq_dep_intro : eq_dep U P p x p x.
```

Proofs above use additional theorems, `eq_dep_eq_sig` state that if two values are dependently equal, then subtypes made of them are equal. The second one, `eq_sig_eq_dep`, states that for types with decidable equality, dependent equality implies equality.

Axiom K

Axiom K (also known as Uniqueness of Identity Proof) is a weaker version of the proof irrelevance axiom. It states that two proofs of the same equality are equal. Thomas Streicher first introduced it in his habilitation thesis [41].

Definition D-2.3.3: Axiom K

```
Definition K := forall (A : Type) (x y : A) (p q : x = y), p = q.
```

Advanced Coq C-2.3.3: Embedding dependent equality in equality

An interesting consequence of using axiom K is that it embeds dependent equality in equality.

```
Fact eq_dep_embedded_in_eq : forall (A : Type) (P : A -> Prop)
  (a : A) (p q : P a), eq_dep A P a p a q -> p = q.
```

Moreover, this fact is equivalent to axiom K. Proofs of these facts are outside the scope of this section, but they can be found in the appendix **Extras/Stricher.v**.

Since axiom K is weaker than the proof irrelevance axiom, we cannot prove using it that every subtype has unique representations. However, we can use it to prove that every quotient-like type (defined in D-2.1.5) has unique representations as defined in D-2.3.1.

Theorem T-2.3.3

```
Theorem quotient_unique_representation : K -> forall (A : Type)
  (f : A -> A) `{normalization A f} (x y : quotient f),
  val x = val y -> x = y.
```

Proof P-2.3.3

```
intros K A f N [x xp] [y yp] H.
cbn in *. subst.
specialize (K _ _ _ xp yp). subst.
reflexivity. Qed.
```

2.3.2 Definitional irrelevance approach

As it was mentioned in this section, we can add the axiom of proof irrelevance to solve the problem of unique representations for every subtype. In Coq, there is another approach to proof irrelevance – SProp.

Advanced Coq C-2.3.4: SProp sort

SProp (Strict Propositions) is a sort of propositions with definitional proof irrelevance as described in [23]. It is still an experimental feature [27] of Coq and requires work to be fully functional. That said, the standard library of Coq has several useful constructions for strict propositions:

Box – a record that lifts SProp proposition to Prop sort.

```
Record Box (A : SProp) : Prop := box { unbox : A }.
```

Squash – an inductive type that squashes proposition into SProp sort making it proof irrelevant.

```
Inductive Squash (A : Type) : SProp :=
  squash : A -> Squash A.
```

sEmpty – an equivalent of False in Prop. From sEmpty any proposition follows.

```
Inductive sEmpty : SProp := .
Definition sEmpty_rect : forall (P : sEmpty -> Type)
  (s : sEmpty), P s.
```

sUnit – an equivalent of True in Prop.

```
Inductive sUnit : SProp := stt : sUnit.
```

Ssig – subtype defined in SProp sort.

```
Record Ssig (A : Type) (P : A -> SProp) : Type :=
  Sexists { Spr1 : A;  Spr2 : P Spr1 }.
```

We can easily prove that subtypes defined using Ssig have unique representations.

Theorem T-2.3.4

```
Theorem Ssig_unique_representation : forall (A : Type)
  (P : A -> SProp) (a b : Ssig P), Spr1 a = Spr1 b -> a = b.
```

Proof P-2.3.4

```
intros A P [a ap] [b bp] H.
cbn in *. subst.
reflexivity. Qed.
```

Using Ssig is very convenient since having unique representations does not require additional axioms. However, the disadvantage of this approach is that the proof that the value satisfies the subtype defining predicate is in the SProp sort. Therefore, using this proof in the Prop sort, where most theorems live, is usually impossible. The only exception is when we can use it to prove sEmpty.

2.3.3 Homotopic approach

If we want to work in the Prop sort without additional axioms, we must find underlying types with unique identity poofs. It is a challenging task. Nevertheless, with help comes homotopy type theory (HoTT) [28]. It is a relatively new branch of type theory focusing on identity proofs. Homotopic interpretation of a type is ω-grupoid. In such groupoid, nodes represent members of a type; paths represent identity proofs; paths between paths represent identity proofs of identity proofs, and so forth.

Homotopy n-types

Homotopy n-types creates a type hierarchy based on the complexity of identity poofs structure (ω-grupoid structure).

Example E-2.3.1: Selected n-types

Developing intuition regarding this concept is important before formally defining the n-th type. Therefore, we will discuss the three most important examples of n-types in this thesis.

`Contr` – it is the lowest minus-second type. The type that lives in `Contr` has exactly one element. A good example of (-2)-type is a `unit`.

```
Class isContr (A : Type) := ContrBuilder {
  center : A;
  contr  : forall x : A, x = center
}.
```

`HProp` – it is the minus-first type. Not to be confused with Coq's `Prop`. The type that lives in it has every element equal to each other. A good example of (-1)-type is `Empty_set`. We can easily prove that each element of the empty type is the same element. However, it has no elements, so it cannot live in `Contr`.

```
Class isHProp (P : Type) :=
  hProp : forall p q : P, p = q.
```

`HSet` – it is the zero type. Not to be confused with Coq's `Set`. The type that lives in it has unique identity proofs. A good example of 0-type is `bool`. Later in this section we will show that it has unique identity proofs. Moreover, it has two elements, so it does not live in `HProp`.

```
Class isHSet (X : Type) :=
  hSet : forall (x y : X) (p q : x = y), p = q.
```

Definition D-2.3.4: n-th type

```
Inductive universe : Type :=
| minus_two   : universe
| S_universe : universe -> universe.

Fixpoint isNType (n : universe) (A : Type) : Type :=
  match n with
  | minus_two     => isContr A
  | S_universe n' => forall x y : A, isNType n' (x = y)
  end.
```

As we see in the definition, n-types are defined by where identity proofs of elements of those types live. The hierarchy starts at the (-2)-type, where only the most primitive contractible types live. Types with identity proofs in `Contr` live in `HProp` since the identity proof type is inhabited for every pair of elements. Therefore, every two elements are equal. Similarly, if identity proofs of type live in `HProp`, then the type lives in `HSet` since one or no identity proof exists for every two elements of an underlying type. The other important fact is that every n-type is also a $(n+1)$-type.

Theorem T-2.3.5: n-type inclusion

```
Theorem NType_inclusion : forall A: Type, forall n : universe,
  isNType n A -> isNType (S_universe n) A.
```

Proof P-2.3.5

Proof of this theorem can be found in the appendix **Lib/HoTT.v**.

Types with decidable equality

If the underlying type of quotient-like type (defined in D-2.1.5) lives in the `HSet` universe, then we can easily prove that it has a unique representations the same way as in Proof P-2.3.3. However, proving that a type lives in `HSet` is not trivial. Fortunately, Hedberg's theorem [24] gives us a valuable sufficiency condition for being `HSet` – decidable equality.

Definition D-2.3.5: Decidable equality

```
Class Decidable (A : Type) :=
  dec : A + (A -> False).

Class DecidableEq (A : Type) :=
  dec_eq : forall x y: A, Decidable (x = y).
```

For a type to have decidable equality means that there exists an algorithm that, in a finite amount of time, decides if two elements of this type are equal.

Theorem T-2.3.6: Hedberg's theorem

```
Theorem hedberg (A : Type) : EqDec A -> isHSet A.
```

Proof P-2.3.6

This proof of Hedberg's theorem is based on the paper [30]. It uses the concept of collapsibility.

```
Class Collapsible (A : Type) := {
  collapse        : A -> A ;
  wconst_collapse : forall x y: A, collapse x = collapse y;
}.
```

It can be easily proven that the type is collapsible if it is decidable.

```
Theorem dec_is_collaps : forall A : Type, Decidable A ->
  Collapsible A.
Proof.
  intros A eq. destruct eq.
  - exists (fun x => a). intros x y. reflexivity.
  - exists (fun x => x); intros x y.
    exfalso; apply f; assumption.
Qed.
```

The same law applies to the type of identity proofs.

```
Class PathCollapsible (A : Type) :=
  path_coll : forall (x y : A), Collapsible (x = y).

Theorem eq_dec_is_path_collaps : forall A : Type,
  DecidableEq A -> PathCollapsible A.
Proof.
  intros A dec x y. apply dec_is_collaps. apply dec.
Qed.
```

We need a simple lemma about the composition of identity proofs.

```
Lemma loop_eq : forall A: Type, forall x y: A, forall p: x = y,
  eq_refl = eq_trans (eq_sym p) p.
Proof.
  intros A x y []. cbn. reflexivity.
Qed.
```

Using the lemma above, we can prove that type with collapsable paths lives in `HSet`.

```
Theorem path_collaps_is_hset (A : Type) : PathCollapsible A ->
  isHSet A.
Proof.
  unfold isHSet, PathCollapsible; intros C x y.
  cut (forall e : x = y, e =
    eq_trans (eq_sym (collapse (eq_refl x))) (collapse e)).
  - intros H p q.
    rewrite (H q), (H p), (wconst_collapse p q).
    reflexivity.
  - intros []. apply loop_eq.
Qed.
```

Composing `dec_is_collaps` with `path_collaps_is_hset`, we get proof that every type with decidable equality lives in `HSet` and has only trivial paths. □

Hedberg's theorem shows that every quotient-like type (defined in D-2.1.5) with an underlying type with decidable equality has unique representations. However, most types with normalization functions (defined in D-1.6.1) have decidable equality.

n-types of function types

It is also worth characterizing n-type of function type based on the n-type of its domain and codomain. To do this, we first need to define homotopic equivalence.

Definition D-2.3.6: Homotopic equivalence

```
Class HEquiv (A B : Type) := {
  izo     : A -> B;
  inv     : B -> A ;
  eisretr : forall b : B, izo (inv b) = b;
  eissect : forall a : A, inv (izo a) = a;
  eisadj  : forall a : A, eisretr (izo a) = ap izo (eissect a);
}.
```

The last element of equivalence `eisadj` is required for the equivalence type to live in `HProp`. This property, however, is not required for this thesis. Therefore, we will skip it, for interested it can be found in [28]. Proof that this is an equivalence relation can be found in the appendix **Lib/HoTT.v**. Since we are working with functions, we need an axiom of functional extensionality.

Definition D-2.3.7: Homotopic functional extensionality

```
Definition happly {A : Type} {B : A -> Type}
  {f g : forall x : A, B x} (p : f = g) (x : A) : f x = g x :=
  match p with
  | eq_refl => eq_refl
  end.

Definition HFunExt : Type := forall (A : Type) (B : A -> Type)
  (f g : forall a : A, B a), IsEquiv (happly (f := f) (g := g)).
```

This definition of functional extensionality is perfect for homotopic reasoning since it uses the homotopic equivalence definition D-2.3.6. Using these tools, we can characterize n-type of function types. The n-type of function type depends solely on the n-type of the codomain.

Theorem T-2.3.7: n-type of function type

```
Theorem fun_is_img_level (A B : Type) (n : universe)
  (funExt : HFunExt) : isNType n B -> isNType n (A -> B).
```

Proof P-2.3.7

For `HProp`, we can prove this theorem.

```
Lemma fun_is_img_hprop (A : Type) (B : A -> Type)
  (funExt : HFunExt) : (forall a : A, isHProp (B a)) ->
  isHProp (forall a : A, B a).
Proof.
  unfold isHProp. intros H f g.
  destruct (F A B f g) as (inv, _, _, _). apply inv.
  intros x; apply H.
Qed.
```

We can also prove this for `Contr`.

```
Lemma fun_is_img_contr (A : Type) (B : A -> Type)
  (funExt : HFunExt) : (forall a : A, isContr (B a)) ->
  isContr (forall a : A, B a).
Proof.
  assert (C' : forall T: Type, T * (isHProp T) -> isContr T).
  { intros T (c, hprop). exists c. intros t. apply hprop. }
  intros H. apply C'. split.
  - intros a. destruct (H a). assumption.
  - apply fun_is_img_hprop; [auto |].
    intros a b b'.
    destruct (H a) as (c, l).
    rewrite (l b), (l b'). auto.
Qed.
```

We need the following theorem about n-type of equivalent types to prove this theorem.

```
Theorem equiv_keep_level (A B : Type) (n : universe) :
   isNType n A -> Equiv B A -> isNType n B.
```

Proof of this lemma is outside of the scope of this section. However, it can be found in the appendix **Lib/HoTT.v**. Now we can prove this fact for dependent functions.

```
Lemma fun_is_img_level_dep (A : Type) (B : A -> Type)
  (n : universe) (funExt : HFunExt) :
  (forall a: A, isNType n (B a)) -> isNType n (forall a: A, B a).
Proof.
  revert A B. induction n; intros A B H; cbn in *.
  - apply fun_is_img_contr; auto.
  - intros f g.
    apply (equiv_keep_level (forall x : A, f x = g x)).
    + apply IHn. intros a. apply H.
    + destruct (funExt A B f g) as (inv, rets, sect, adj).
      econstructor; econstructor; apply adj.
Qed.
```

Since our theorem is a particular case of the above lemma, we consider the theorem to be proved. □

Characterization of dependent pair equality

Advanced Coq C-2.3.5: Dependent pair

In Coq, the type of *dependant pair* is defined using inductive type.

```
Inductive sigT (A : Type) (P : A -> Type) : Type :=
  existT : forall x : A, P x -> sigT A P.
```

`{x : A & P x}` is the notation for dependent pairs.

Subtypes are based on dependent pairs. Therefore, knowing how the equality of dependent pairs can be characterized is beneficial. In the case of simple non-dependent pairs, the characterization is straightforward.

Theorem T-2.3.8: Characterization of pair equality

```
Theorem pair_eq : forall (A B : Type) (a x : A) (b y : B),
  (a, b) = (x, y) -> a = x /\ b = y.
```

Proof P-2.3.8

```
intros. inversion H. split; trivial. Qed.
```

Using the same definition for dependent pairs is impossible. The first issue is the error of ill-defined equality since `P x` and `P y` differ in the Coq type system, even if `x = y`, and elements of two different types cannot be in equality relation. Even if we change our problem to the characterization of equality for dependent pairs with the same first element, it is still impossible to characterize it using simple equality. Such equality can be characterized using dependent equality known from Fact C-2.3.2. Moreover, that characterization using simple equality is correct only if dependent equality is equivalent to equality. From Fact C-2.3.3, we know that it is true if and only if axiom K is true. To characterize the general case of dependent pairs equality, we must first define the transport along an equality path.

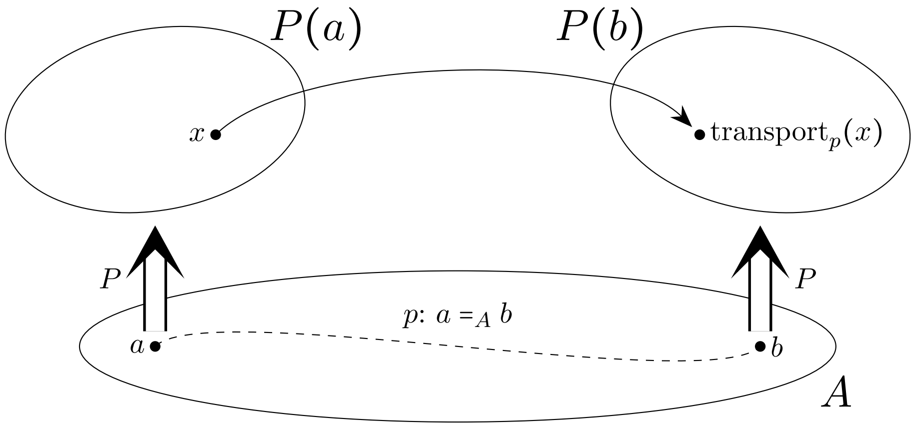
Definition D-2.3.8: Transport

```
Definition transport {A : Type} {a b : A} {P : A -> Type}
  (p : a = b) (x : P a) : P b :=
match p with
| eq_refl => x
end.
```

Transport construction moves an element of type `P a` to a new type `P b` along the path (proof of equality) `p : a = b`. Using this construction, we can characterize equality of dependent pairs.

Visualization V-2.3.1: Transport

P is a family of types indexed by elements of type A. Two elements, a and b, are equal in A, and p is a proof of this fact (path). Element x of type $P(a)$ can be transported along this proof (path) to type $P(b)$.



Theorem T-2.3.9: Characterization of dependent pair equality

```
Theorem dep_pair_eq : forall (A : Type) (P : A -> Type)
  (x y : A) (p : P x) (q : P y), existT P x p = existT P y q ->
  exists e : x = y, transport e p = q.
```

Proof P-2.3.9

```
intros A P x y p q H. inversion H.
exists eq_refl. cbn. trivial. Qed.
```

Chapter 3

Quotient types utilizing subtyping

The previous chapter discusses using subtyping to define quotient-like types in Coq. This chapter lists examples of quotients that can be easily defined in this way. It focuses on defining quotients with an underlying type living in `HSet`. Therefore, we are able to use the classical equality definition and work in `Prop` sort.

3.1 Unordered pairs

Definition D-3.1.1: Unordered pairs

In mathematics, an *unordered pair* is a set of the form $\{a, b\}$, where $\{a, b\} = \{b, a\}$ [36]. In contrast to the ordered pair where $(a, b) \neq (b, a)$.

As mentioned in the introduction, there is no normalization function for unordered pairs in the most general case. Therefore, we will only discuss the case when the underlying type has a full decidable order (defined in D-3.1.2). Antisymmetry is required to prove the uniqueness of representations and universality (fullness) is required to define a computable function to construct an unordered pair out of two elements.

Definition D-3.1.2: Full decidable order

```
Class FullOrd (A : Type) := {
  ord  : A -> A -> bool;
  asym : forall x y : A, ord x y = true -> x = y;
  full : forall x y : A, ord x y = true \/ ord y x = true;
}.
```

Definition D-3.1.3: Unordered pairs in Coq

Unordered pairs can be defined as a triple of first and second element and a proof that states they are in the correct order.

```
Record UPair (A : Type) `{FullOrd A} := {
  fst    : A;
  snd    : A;
  sorted : ord fst snd = true;
}.
```

In order to use such defined unordered pairs, we need to define a constructor.

Function F-3.1.1: Constructor of unordered pair

```
Definition make_UPait {A : Type} `{FullOrd A} (x y : A) : UPair A.
Proof.
  destruct (ord x y) eqn:o.
  - econstructor. apply o.
  - assert (o1: ord y x = true) by
    (destruct (full x y); [rewrite H0 in o; discriminate | auto]).
    econstructor. apply o1.
Qed.
```

3.1.1 Uniquness of representations

To talk about the uniqueness of representations, firstly we need to define the equivalence relation defining this quotient construction. In the case of unordered pairs, if two pairs contain the same elements, they are in a quotient-defining relation.

Definition D-3.1.4: Equivalence relation for unordered pairs

```
Definition contains {A : Type} `{FullOrd A}  (x y : A)
  (p : UPair A) := (fst p = x /\ snd p = y) \/
    (fst p = y /\ snd p = x).

Definition UPair_equiv {A : Type} `{FullOrd A} (p q : UPair A) :=
  forall x y : A, contains x y p <-> contains x y q.
```

Having a formal definition of equivalence relation, we can formally verify the claim of unique representations for such defined unordered pairs.

Theorem T-3.1.1: Uniquness of representation for unordered pairs

```
Theorem UPair_uniq (A : Type) `{FullOrd A} (p q : UPair A)
  (x y : A) : contains x y p -> contains x y q -> p = q.
```

Proof P-3.1.1

```
intros [(cp1 & cp2) | (cp1 & cp2)] [(cq1 & cq2) | (cq1 & cq2)];
destruct p, q; cbn in *; subst.
2-3: assert (x = y) by (apply asym; assumption); subst.
1-4: f_equal; apply bool_is_hset. Qed.
```

As we see, the last part of this proof uses `bool_is_hset` theorem. As we know, `bool` has decidable equality. According to Hedberg's theorem T-2.3.6, we know it also has unique identity proofs. Formal proof can be found in the appendix **Lib/HoTT.v**.

3.2 Finite multisets

Definition D-3.2.1: Multiset

In mathematics, a *multiset* (also known as *bag* or *mset*) is a modification of a set concept that allows for multiple instances of the same element [36].

A multiset is another example of a quotient type with applications in everyday programming [39]. It is a data structure known from mathematics where the order of elements does not matter. It can be considered as unordered list. Similar to unordered pairs, there is no normalization function for multisets in the most general case. Therefore, in this thesis we will discuss only multisets with underlying types with decidable linear order.

Definition D-3.2.2: Decidable linear order

```
Class LinearOrder {A : Type} := {
  ord      : A -> A -> bool;
  anti_sym : forall x y : A, ord x y = true -> ord y x = true ->
               x = y;
  trans    : forall x y z : A, ord x y = true ->
               ord y z = true -> ord x z = true;
  full      : forall x y : A, ord x y = true \/ ord y x = true;
}.
```

3.2.1 The equivalence relation

Two multisets are in equivalence relation when they contain the same elements in the same quantity. In other words, when they both are permutations of the same list of elements. We propose using an alternative permutation definition.

Definition D-3.2.3: Permutaion

```
Fixpoint count {A : Type} (p : A -> bool) (l : list A) : nat :=
  match l with
  | nil      => 0
  | cons h t => if p h then S (count p t) else count p t
  end.

Definition permutation {A : Type} (a b : list A) :=
  forall p : A -> bool, count p a = count p b.
```

This definition works for types with decidable equality, and as we know, decidable linear order implies decidable equality. For such types, the special case of a decidable predicate is the predicate being equal to a specific element. This definition does not require defining all laws of permutations. Therefore, it is easier to use in the context of a sorting function. Proof of the fact that this definition is equivalent to the classic permutation definition for types with decidable equality can be found in the appendix **Extras/Permutations.v**.

3.2.2 The normalization function

We can use the sorting function as a normalization function for a multiset quotient type. One of the sorting functions with the best asymptotic complexity is merge sort. Moreover, it has a nice functional definition, making it the ideal candidate for a multiset normalization function. We propose a definition using binary trees.

Definition D-3.2.4: Binary tree

```
Inductive BT (A : Type) : Type :=
| leaf : A -> BT A
| node : BT A -> BT A -> BT A.
```

We also define the insert function that keeps the tree balanced out and the function that converts a list into a balanced-out tree.

Function F-3.2.1: Conversion to balanced-out tree

```
Fixpoint BTInsert {A : Type} (x : A) (tree : BT A) :=
  match tree with
  | leaf y   => node (leaf x)(leaf y)
  | node l r => node r (BTInsert x l)
  end.

Fixpoint listToBT {A : Type} (x : A) (ls : list A) : BT A :=
  match ls with
  | []        => leaf x
  | l :: ls' => BTInsert x (listToBT l ls')
  end.
```

As expected, we also need a merging function for two sorted lists.

Function F-3.2.2: Merge

```
Fixpoint merge {A : Type} (ord : A -> A -> bool) (l1 : list A)
  : (list A) -> list A :=
  match l1 with
  | []        => fun (l2 : list A) => l2
  | h1 :: t1 => fix anc (l2 : list A) : list A :=
    match l2 with
    | []        => l1
    | h2 :: t2 => if ord h1 h2
                  then h1 :: (merge ord t1) l2
                  else h2 :: anc t2
    end
  end.
```

Now we can define the whole merge sort function, using the defined above ancillary functions.

Function F-3.2.3: Merge sort

```
Fixpoint BTSort {A : Type} (ord : A -> A -> bool)
  (t : BT A) : list A :=
  match t with
  | leaf x   => [x]
  | node l r => merge ord (BTSort ord l) (BTSort ord r)
  end.
```

```
Definition mergeSort {A : Type} `{LinearOrder A}
  (l : list A) : list A :=
  match l with
  | []      => []
  | x :: l' => BTSort ord (listToBT x l')
  end.
```

3.2.3 Uniqueness of representations

Theorem T-3.2.1: Uniquness of representation for multisets

For every multiset, there is exactly one element in `quotient mergeSort` type (defined in D-2.1.5).

Proof P-3.2.1

A sorting function should satisfy two requirements:

- the output is a sorted list,
- the output is a permutation of input.

Proofs that the sorting function satisfies the above requirements can be found in the appendix **Lib/MergeSort.v**.
Moreover, we can show that for every list of elements, only one permutation of this list is sorted. Proof of this fact is in the appendix **Lib/Sorted.v**.
Combining those facts, we know that sorting is an idempotent function, and there is only one output of this function for each equivalence class. More formal proof can be found in the appendix **FiniteMultiSet.v**. □

3.3 Finite sets

Sets are a basic construction known from mathematics [36]. It is important to mention that a particular element may appear in a set at most once. Therefore, adding an element to a set that already contains this element does not change the result. Moreover, the order of elements in sets does not matter. Sets can be considered as sorted and deduplicated lists. Like the two previous examples, sets do not have a normalization function in the most general case. Therefore, we will discuss only sets with underlying types with decidable linear order (defined in D-3.2.2).

3.3.1 The equivalence relation

Two sets are in equivalence relation when they contain the same elements. In other words, for every element in underlying type, if it is in the first set, it has at least one copy in the second set. Here we also propose an alternative definition of this relation similar to the previous definition of permutations (defined in D-3.2.3).

Definition D-3.3.1: Set equivalence

```
Fixpoint any {A : Type} (p : A -> bool) (l : list A) : bool :=
  match l with
  | []        => false
  | (x :: l') => if p x then true else any p l'
  end.

Definition Elem_eq {A : Type} (l l' : list A) : Prop :=
  forall p : A -> bool, any p l = any p l'.
```

For types with decidable equality, the definition above is equivalent to the classical definition. Proof of this fact can be found in the appendix **Lib/Deduplicated**.

3.3.2 The normalization function

For sets, the normalization function needs to remove duplicates and sort elements. A common approach for such a computer science problem is binary tree [29]. This paper also uses it to define a normalization function for a set.

Definition D-3.3.2: Binary tree

```
Inductive tree (A : Type) : Type :=
| leaf : tree A
| node : A -> tree A -> tree A -> tree A.
```

In order to use the insert function, we need to use a more convenient interface for comparing elements.

Function F-3.3.1

```
Definition comp {A : Type} `{LinearOrder A} (x y : A) : comparison
  := if ord x y then (if ord y x then Eq else Gt) else Lt.
```

We can define a function that inserts elements into the sorted binary tree if and only if the tree does not already contain them.

Function F-3.3.2: Insertion to binary tree

```
Fixpoint insert_tree {A : Type} `{LinearOrder A} (x : A)
  (t : tree A) : tree A :=
  match t with
  | leaf       => node x leaf leaf
  | node v l r => match comp x v with
                  | Lt => node v (insert_tree x l) r
                  | Eq => node v l r
                  | Gt => node v l (insert_tree x r)
                  end
  end.
```

By converting a list into a sorted and deduplicated tree and back to a list, we get a sorted and deduplicated list.

Function F-3.3.3: Deduplicating sort

```
Fixpoint to_tree {A : Type} `{LinearOrder A} (l : list A)
  : tree A :=
  match l with
  | []      => leaf
  | x :: l' => insert_tree x (to_tree l')
  end.

Fixpoint to_list {A : Type} (l : tree A) : list A :=
  match l with
  | leaf       => []
  | node x l r => to_list l ++ [x] ++ to_list r
  end.

Definition DSort {A : Type} `{LinearOrder A} (l : list A)
  : list A := to_list (to_tree l).
```

3.3.3 Uniqueness of representations

Theorem T-3.3.1: Uniqueness of representations for sets

For every set, there is exactly one element in `quotient DSort` type (defined in D-2.1.5).

Proof P-3.3.1

A sorting and deduplicating function should satisfy the following requirements:

- the output is a deduplicated list,
- the output is a sorted list,
- the output and input contain the same elements.

Proofs that the function above meets those requirements can be found in the appendix **Lib/DedupSort.v**.

In addition, we know that each set has exactly one sorted and deduplicated list. Proof of this is in the appendix **Lib/Deduplicated.v**.

On the basis of those facts we can conclude that the normalization function of a set is idempotent. There is exactly one element for each set defining equivalence class in the codomain of this function. Formal proof can be found in the appendix **FiniteSet.v**. □

Chapter 4

Quotient types as normalization traces

This chapter discusses quotient types defined as widely understood traces of normalization functions. The inductive types presented here are inspired by how normalization works for selected quotient types. Some examples reinvented in this chapter are commonly used in Coq, and some are new constructions created for this thesis.

4.1 Free monoids

Definition D-4.1.1: Monoid

Monoid is an algebraic structure $(\mathbf{A}, \circ)$, where $\mathbf{A}$ is a carrier set and $\circ : \mathbf{A} \to \mathbf{A} \to \mathbf{A}$ is a binary operation and following requirements are fulfilled:

- there exists a neutral element $\mathrm{e} \in \mathbf{A}$, such that $\forall x \in \mathbf{A}.\ e \circ x = x = x \circ e$,
- the binary operation $\circ : \mathbf{A} \to \mathbf{A} \to \mathbf{A}$ is associative, that means $\forall x, y, z \in \mathbf{A}.\ x \circ (y \circ z) = (x \circ y) \circ z$.

Example E-4.1.1: Monoid of single argument functions

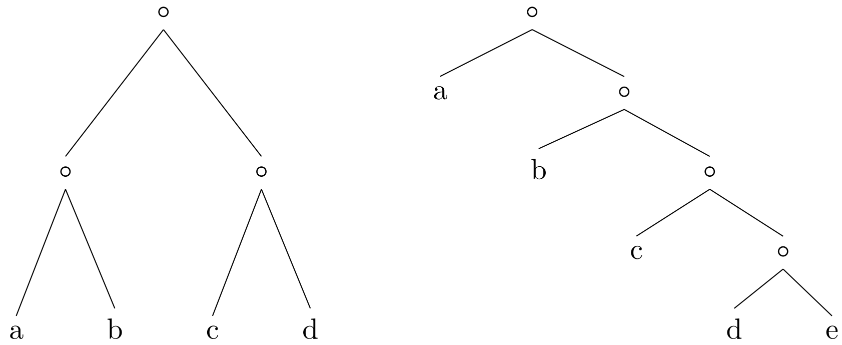
An algebraic structure of single argument functions $(A \to A)$ with function composition operation is an example of a monoid. Function composition is an associative operation, and identity function is a neutral element of this monoid.

Definition D-4.1.2: Free object

Free object $\mathbf{A}$ over algebraic structure is the object where the only equations that hold between elements of the free object are those that follow from the axioms defining this algebraic structure [22].

Visualization V-4.1.1: Equivalent elements of a monoid

Two equivalent notations of $a \circ b \circ c \circ d$.



Example E-4.1.2: Free monoid

Lists of natural numbers are a great example of free monoids. The type of list is often called a type of free monoid. The binary operation in this monoid is the concatenation of lists (++). Concatenation is associative - the order of concatenation operation does not matter. The empty list is the neutral element in this free monoid.

```
[1; 2] ++ [3] = [1] ++ ([2] ++ [3]) = [1] ++ [] ++ [2; 3]
```

4.1.1 A naive representation

We are accustomed to list representation of free monoids. Elements of a free monoid we call *strings*. However, not every notation of a string can be expressed as a unique list since, every notation of the same string shares the same list representation. We call the type of every notation of elements of free monoid a naive representation. A first attempt to implement the type of free monoids in Coq for someone unfamiliar with this concept would likely look similar to the definition below.

Definition D-4.1.3: Naive free monoid representation

```
Inductive FreeMonoid (A : Type) :=
| leaf : FreeMonoid A
| var  : A -> FreeMonoid A
| op   : FreeMonoid A -> FreeMonoid A -> FreeMonoid A.
```

As we can see in the definition above, we have three constructors: an empty string constructor `leaf`, a singleton constructor `var`, and a string concatenation constructor `op`.

4.1.2 The normalized representation

As it is easy to notice, naive representation D-4.1.3 is not unique for every string. For example, there are infinitely many representations of the neutral element: `leaf`, `op leaf leaf` etc. To deal with this problem, we must define a normalized form of strings:

$$(a_1 \circ (a_2 \circ (a_3 \circ \cdots \circ (a_n \circ e) \dots))$$

As expected, we devised a classic list representation of free monoid.

Definition D-4.1.4: List as free monoid

```
Inductive list (A : Type) :=
| nil  : list A
| cons : A -> list A -> list A.
```

The normalization function finds the first element in the naively defined string, puts it as the first constructor, and recursively normalizes the rest of the string. The normalization function returns a natural element if the string is equivalent to a neutral element. The formal definition of normalization function is not very educative. Therefore, we will skip it. However, we will present a conversion function from a naive representation to a normalized one.

Function F-4.1.1: Conversion from a naive representation to a list

```
Fixpoint free_to_list {A : Type} (m : FreeMonoid A) : list A :=
  match m with
  | leaf   => []
  | var x  => [x]
  | op x y => to_list x ++ to_list y
  end.
```

4.2 Integers

Integers are another classic quotient type with many applications in computer science. They result from extending natural numbers to an additive group by adding an opposite element to every natural number. They can be naively represented as a pair of two natural numbers. The first number represents the number of predecessors, and the second represents the number of successors applied to zero.

4.2.1 A naive representation

Definition D-4.2.1: Naive integers representation

```
Definition Int : Type := nat * nat.
```

For this representation, we can easily define the addition function by adding the number of predecessors and successors of two numbers.

Function F-4.2.1: Addition of naive integers

```
Definition int_add (n m : Int) : Int :=
  let (a, b) := n in let (c, d) := m in (a + c, b + d).
```

Despite many advantages, this representation is rarely used due to a lack of unique representations. As it is easy to notice, there are infinitely many representations of zero: $1 + (-1)$, $2 + (-2)$, etc. Therefore, we can define a normalization function that will remove redundant pairs of successor and predecessor. After normalization, an integer is made out of either predecessors or successors.

Function F-4.2.2: Normalization of naive integers

```
Fixpoint int_norm' (x y : nat) : (nat * nat) :=
  match x, y with
  | S x', S y' => int_norm' x' y'
  | _, _       => (x, y)
  end.

Definition int_norm (p : Int) : Int :=
  let (x, y) := p in int_norm' x y.
```

4.2.2 The normalized representation

Having a normalization function for our naive representation, we can, based on this normalization process, define an inductive type of normalized representation. The trace of this normalization process is not very interesting until the last recursion. The length of this process depends entirely on how far from the normalized representation the number is. The last step of the normalization process is in function F-4.2.2 simplified to `_, _`. However, we can expand `_, _` to three distinct cases:

`S x', O` – for negative `S x'`,

`O, S y'` – for positive `S y'`,

`O, O` – for zero.

Using this trace, we can define the following inductive type for integers. It is a representation known from Coq's standard library [27].

Definition D-4.2.2: Normalized integers representation

```
Inductive Z : Type :=
| Pos  : nat -> Z
| Zero : Z
| Neg  : nat -> Z.
```

We can also modify a normalization function F-4.2.2 to convert naive representations into normalized ones.

Function F-4.2.3: Integers conversion

```
Fixpoint int_to_Z (x y : nat) : Z :=
  match x, y with
  | S x', S y' => int_to_Z x' y'
  | S x', O    => Neg x'
  | O   , S y' => Pos y'
  | O   , O    => Zero
  end.
```

4.2.3 Basic operations

We define some basic operations for the normalized representation of integers D-4.2.2. Like natural numbers, starting with successor and predecessor definitions is the best option.

Function F-4.2.4: Successor

```
Definition succ (n : Z) : Z :=
  match n with
  | Pos k     => Pos (S k)
  | Zero      => Pos O
  | Neg O     => Zero
  | Neg (S n) => Neg n
  end.
```

Function F-4.2.5: Predecessor

```
Definition pred (n : Z) : Z :=
  match n with
  | Pos (S n) => Pos n
  | Pos O     => Zero
  | Zero      => Neg O
  | Neg n     => Neg (S n)
  end.
```

Both definitions are straightforward but substantially more complicated than the same definitions of operations for naive representations. The fourth case was required to handle the transition to zero in both cases.

Function F-4.2.6: Negation

```
Definition neg (n : Z) : Z :=
  match n with
  | Pos k => Neg k
  | Zero  => Zero
  | Neg k => Pos k
  end.
```

The definition of negation is simple and does not require a discussion. The next basic operation for integers that we discuss is addition.

Function F-4.2.7: Map n times

```
Fixpoint map_n {A : Type} (n : nat) (f : A -> A) (x : A) : A :=
  match n with
  | O    => x
  | S n' => f (map_n n' f x)
  end.
```

Function F-4.2.8: Addition

```
Definition add (a b : Z) : Z :=
  match a with
  | Pos n => map_n (S n) succ b
  | Zero  => b
  | Neg n => map_n (S n) pred b
  end.
```

For defining addition, we first need to define an ancillary function that applies some function n times. Using it, we can define addition as applying the successor or predecessor operation several times. Subtraction can be defined as the addition of an opposite number.

Function F-4.2.9: Multiplication

```
Definition mul (a b : Z) : Z :=
  match a with
  | Pos n => map_n (S n) (add b) Zero
  | Zero  => Zero
  | Neg n => neg (map_n (S n) (add b) Zero)
  end.
```

The multiplication operation can be defined as adding the same number several times. As we know, the addition and multiplication of integers are commutative and associative. Moreover, multiplication is distributive over addition. Proofs of those facts are too long to be presented in this paper. Though, they are in the appendix **Integers.v**.

4.3 Exotic integers

Integers representation defined in the previous section is often used in Coq due to its symmetric simple definition and computational properties. For example, the addition has linear complexity to one of the arguments. A small unsatisfactory detail is that we used the inductive type of natural numbers in the definition of integers. The question arises if there is a definition of integers that uses a single inductive type.

4.3.1 The alternative normalized representation

We can define the normalization process in an alternative way. Instead of finishing when we determine if the number is positive or negative, we can continue the process by moving successors from input to output. Of course, this step is de facto an identity function on natural numbers with linear complexity.

Function F-4.3.1: Alternative normalization

```
Fixpoint alt_norm (n m : nat) : nat * nat :=
  match n, m with
  | S n', S m' => alt_norm n' m'
  | O   , S _  => map_n n
      (fun (i : nat * nat) => let (x, y) := i in (S x, y)) (O, O)
  | O   , _    => map_n m
      (fun (i : nat * nat) => let (x, y) := i in (x, S y)) (O, O)
  end.
```

This unnecessary step changes the trace of the normalization process. Now, after determining the sign of the integer, the length of the trace defines its value. This part of the trace is almost identical for positive and negative numbers. Therefore, they will both use a single constructor `Next`. We will use two different nullary constructors for parts of traces where the integer sign is determinated, named `Zero` and `MinusOne`.

Definition D-4.3.1: Exotic integers representation

```
Inductive Z' : Type :=
| Zero     : Z'
| MinusOne : Z'
| Next     : Z' -> Z'.
```

The `Next` constructor is both successor and predecessor depending on the integer to which it is applied. A negative integer is a predecessor; on a zero or positive number, it is a successor. This representation is uncommon. We were not able to find any paper or document in which it was used. This is most likely because the definition from the previous section meets almost all criteria for perfect integer representation.

Function F-4.3.2: Conversion to exotic integrs

```
Fixpoint to_Z' (n m: nat) : Z' :=
  match n, m with
  | S n', S m' => to_Z' n' m'
  | O   , S m' => map_n m' Next MinusOne
  | _   , O    => map_n n Next Zero
  end.
```

4.3.2 Basic operations

This definition of integers, however, could be more convenient for defining operations. As we can notice, to determine the sign of a number, we need to check the last constructor. Since our definition uses the same construction for successor and predecessor, even a simple operation of computing the successor of a number has linear complexity.

Function F-4.3.3: Successor

```
Fixpoint succ (k : Z') : Z' :=
  match k with
  | Zero           => Next Zero
  | MinusOne       => Zero
  | Next Zero      => Next (Next Zero)
  | Next MinusOne  => MinusOne
  | Next k'        => Next (succ k')
  end.
```

Function F-4.3.4: Predecessor

```
Fixpoint pred (k : Z') : Z' :=
  match k with
  | Zero           => MinusOne
  | MinusOne       => Next MinusOne
  | Next Zero      => Zero
  | Next MinusOne  => Next (Next MinusOne)
  | Next k'        => Next (pred k')
  end.
```

Because of those difficulties, we recommend using an intermediate form of naively defined integers D-4.2.1 or the previous normalized definition of integers representation D-4.2.2 to carry out computations. The conversion functions can be easily defined. The proof of standard and exotic integers definitions being isomorphic can be found in the appendix **ExoticInteger.v**.

4.4 Positive rational numbers

In contrast to integers, coming up with a type for rational numbers with unique representation is a challenging task. This section is based on the paper of Yves Bertot [10], which presents a solution to this problem based on a trace of the Euclidean algorithm.

4.4.1 A naive representation

A naive representation of positive rational numbers is identical to that of integers. It comprises two natural numbers, one for a numerator and one for a denominator. Of course, we cannot divide by zero. Therefore, we can interpret the second number as predecessor of the denominator or accept that some pairs do not represent any number.

Definition D-4.4.1: Naive representation of rational numbers

```
Definition Q' : Type := nat * nat.
```

We chose the second option since we want to use an already defined coq operation on natural numbers. As we know from mathematics, there are many notations for the same fraction. $\frac{1}{2}$ or $\frac{2}{4}$ represents the same rational number. Every fraction can be reduced to an irreducible fraction. In order to find this irreducible fraction, we need to divide the numerator and denominator by their greatest common divisor. There are many algorithms to compute this value, but one of the simplest is the Euclidean algorithm.

Function F-4.4.1: Euclidean algorithm

```
Fixpoint euclid (p q : nat) : nat :=
  match compare p q with
  | Eq => p
  | Gt => euclid (p - q) q n'
  | Lt => euclid p (q - p) n'
  end.
```

The Euclidean algorithm is based on the law that states $\gcd(a,b) = \gcd(a-b,b)$ if $a > b$. The observation that will be important for defining a normalized representation is that the trace of this algorithm does not change if both numbers are multiplied by the same positive natural number, as in the example shown below.

$$\begin{aligned} 11k &> 5k \\ (11-5)k = 6k &> 5k \\ (6-5)k = k &< 4k \\ k &< 3k = (4-1)k \\ k &< 2k = (3-1)k \\ k &= k = (2-1)k \end{aligned} \tag{4.1}$$

The trace, defined as inequality signs between values, does not depend on the chosen value k. The other observation is that we can compute the input values by knowing the trace of the Euclidean algorithm and the output. Without the output, we can still compute the proportions of input values. Proofs of those statements can be found in the appendix **Qplus.v**.

4.4.2 The normalized representation

Based on those observations, we define an inductive type for positive rational numbers. It has three constructors: `N` when the nominator is greater, `D` when the denominator is greater, and `One` when both values are the same.

Definition D-4.4.2: Normalized representation

```
Inductive Qplus : Type :=
| One : Qplus
| N   : Qplus -> Qplus
| D   : Qplus -> Qplus.
```

This construction also provides a nice induction rule for positive rational numbers. The type `Qplus` is based on the Euclidean algorithm. Therefore, the conversion function is a slightly modified version of the Euclidean algorithm that returns its trace instead of computing the greatest common divisor.

Function F-4.4.2: Conversion to Euclidean algorithm trace

```
Fixpoint qplus_c' (p q n : nat) : Qplus :=
  match n with
  | O    => One
  | S n' => match compare p q with
            | Eq => One
            | Gt => N (qplus_c' (p - q) q n')
            | Lt => D (qplus_c' p (q - p) n')
            end
  end.

Definition qplus_c (x : Q') : Qplus :=
  let (p, q) := x in qplus_c' p q ((p + q) / gcd p q).
```

This function differs from the Euclidean algorithm defined in F-4.4.1 because Coq's termination checker cannot deduce the decreasing argument. Therefore, defining the Euclidean algorithm this way is invalid in Coq (8.14.1). We circumvent this problem by adding an argument (the fuel) that constantly decreases, satisfying the termination condition. In order to not change the algorithm output, we need to set the fuel to be big enough. We know that the Euclidean algorithm needs fewer steps to compute the result than the sum of its inputs. Moreover, as we noticed, multiplying inputs by positive numbers does not change the computations so that we can divide the fuel by the greatest common divisor of inputs.

Function F-4.4.3: Conversion to naive representation

```
Fixpoint qplus_i (x : Qplus) : Q' :=
  match x with
  | One  => (1, 1)
  | N x' => let (p, q) := qplus_i x' in (p + q, q)
  | D x' => let (p, q) := qplus_i x' in (p, p + q)
  end.
```

The inverse conversion function is an even simpler construction. It is based on $(p - q) + q = p$. Of course, the inverse function does not return the same fraction but the equivalent irreducible fraction. The proof that these functions are inversions up to equivalency can be found in the appendix **Qplus.v**

4.4.3 Extension to the field of rational numbers

There is a natural extension to the field of rational numbers. It uses the same construction as the extension of natural numbers to integers (see D-4.2.2).

Definition D-4.4.3: Type of normalized rational numbers

```
Inductive FullQ :=
| QPos  : Qplus -> FullQ
| QZero : FullQ
| QNeg  : Qplus -> FullQ.
```

This type uses defined in this section traces of Euclidean algorithm as the normalized representation of a positive rational number. Therefore, we can extend most of the theorems defined for positive rational numbers to the field extension.

4.4.4 Basic operations

In the field of rational numbers, we can define basic operations. However, the proposed normalized representation is not suited for computations. Therefore, we will perform computations in the homomorphic representation using a pair of two integers. However, only a fraction with a positive denominator represents a valid rational number.

Definition D-4.4.4: Type of naive rational numbers

```
Definition Q : Type := Z * Z.
```

In order to use the simpler representation to perform computations, we need to define conversion functions.

Function F-4.4.4: Absolute value

```
Definition abs' (n : Z) : nat :=
  match n with
  | Pos k => S k
  | Zero  => O
  | Neg k => S k
  end.
```

Function F-4.4.5: Embedding natural numbers into integers

```
Definition toPos (x : nat) : Z :=
  match x with
  | O   => Zero
  | S n => Pos n
  end.
```

We first need an absolute value function F-4.4.4 and another function F-4.4.5 that will transform natural number to integer.

Function F-4.4.6: Conversion to the normalized rational number

```
Definition fullQ_c (q : Q') : FullQ :=
  match q with
  | (Pos n, d) => QPos (qplus_c (S n, abs' d))
  | (Zero , d) => QZero
  | (Neg n, d) => QNeg (qplus_c (S n, abs' d))
  end.
```

The conversion function F-4.4.6 works only for fractions with a positive denominator.

Function F-4.4.7: Conversion to the naive rational number

```
Definition fullQ_i (q : FullQ) : Q :=
  match q with
  | QPos q' => let (n, d) := qplus_i q' in (toPos n, toPos d)
  | QZero   => (Zero, Pos O)
  | QNeg q' => let (n, d) := qplus_i q' in (-toPos n, toPos d)
  end.
```

The inverse conversion function F-4.4.7 produces an irreducible fraction with a positive denominator. Having all conversion functions, we can define the addition operation.

Function F-4.4.8: Addition

```
Definition Qadd' (x y : Q') : Q' :=
  match x, y with
  | (n, d), (n', d') => (d' * n + d * n', d * d')
  end.

Definition Qadd (x y : FullQ) : FullQ :=
  fullQ_c (Qadd'(fullQ_i x) (fullQ_i y)).
```

The additional operation adds numerators of two fractions reduced to a common denominator.

Function F-4.4.9: Negation

```
Definition Qneg (q : FullQ) : FullQ :=
  match q with
  | QPos x => QNeg x
  | QZero  => QZero
  | QNeg x => QPos x
  end.
```

The negation switches the sign of a rational number. Subtraction can be defined as the addition of a negative number.

Function F-4.4.10: Multiplication

```
Definition Qmul' (x y : Q') : Q' :=
  match x, y with
  | (n, d), (n', d') => (n * n', d * d')
  end.

Definition Qmul (x y : FullQ) : FullQ :=
  fullQ_c (Qmul' (fullQ_i x) (fullQ_i y)).
```

The multiplication operation multiplies both numerators and denominators.

Function F-4.4.11: Inversion

```
Fixpoint Qplus_inv (q : Qplus) : Qplus :=
  match q with
  | N x => D (Qplus_inv x)
  | One => One
  | D x => N (Qplus_inv x)
  end.

Definition Qinv (q : FullQ) : FullQ :=
  match q with
  | QPos x => QPos (Qplus_inv x)
  | QZero  => QZero
  | QNeg x => QNeg (Qplus_inv x)
  end.
```

The inversion can be easily computed for the Euclidean algorithm trace by swapping two constructors `N` and `D`. As we know, the inverse operation switches the numerator and the denominator. By swapping inputs of the Euclidean algorithm, we changed which number was greater at each step. The division can be defined as multiplying the inversion. The proof that rational numbers with those operations are an algebraic field can be found in the appendix **Extras/FullQ.v**.

4.5 Free groups

In this chapter, we have already learned how to define a free monoid. In this section, we define another free algebraic structure – free group. Elements of a free group we will call *words*. The construction presented in this subsetion was created for this thesis. The definition of groups and free groups is a common problem in Coq [13] [38]. However, no similar definitions have been found in any paper.

Definition D-4.5.1: Group

An algebraic structure $(\mathbf{A}, \circ)$, where $\mathbf{A}$ is a carrier set and $\circ : \mathbf{A} \to \mathbf{A} \to \mathbf{A}$ is a binary operation is a *group* [22] if:

- there exists a neutral element $\mathrm{e} \in \mathbf{A}$, such that $\forall x \in \mathbf{A}.\ e \circ x = x = x \circ e$.
- the binary operation $\circ : \mathbf{A} \to \mathbf{A} \to \mathbf{A}$ is associative, that means $\forall x, y, z \in \mathbf{A}.\ x \circ (y \circ z) = (x \circ y) \circ z$.
- for every element x there is an inverse element x^{-1}, such that $x \circ x^{-1} = x^{-1} \circ x = e$.

Example E-4.5.1: Group of permutations

An interesting example is a group of permutations S_n, with permutations composition as a binary operation. The composition of the function (permutation is a function), as we learned in the example of monoid E-4.1.1, is associative. The constant permutation is the neutral element. We can compute inverse permutation by swapping inputs with outputs for every permutation.

$$\begin{pmatrix} 1 & 2 & 3 & 4 & 5 \\ 2 & 3 & 1 & 5 & 4 \end{pmatrix}^{-1} = \begin{pmatrix} 1 & 2 & 3 & 4 & 5 \\ 3 & 1 & 2 & 5 & 4 \end{pmatrix}$$

4.5.1 A naive representation

We know that the type of free monoids can be represented as the type of lists. The difference between a monoid and a group is that in the group there are inverse elements and in monoid there are not. Therefore, we can naively represent the type of free groups as a type of lists of elements and their signs (information if the element is inverted or not).

Definition D-4.5.2: Naive free group representation

```
Definition NaiveFreeGroup (A : Type) := list (bool * A).
```

As it is easy to notice, this representation is not unique for every element of a free group. Therefore, we need to define a normalization function. In our case, we will remove two elements next to each other with opposite signs.

Definition D-4.5.3: Normalized naive representation

```
Inductive Normalized {A : Type} : NaiveFreeGroup A -> Prop :=
| NNil   : Normalized []
| NSingl : forall (b : bool) (v : A), Normalized [(b, v)]
| NCons  : forall (b b' : bool) (v w : A) (x : NaiveFreeGroup A),
             v <> w \/ b = b' -> Normalized ((b', w) :: x) ->
             Normalized ((b, v) :: (b', w) :: x).
```

We need an underlying type with decidable equality to define the normalization function.

Definition D-4.5.4: Decidable equality in Coq

```
Class EqDec (A : Type) := {
  eqf         : A -> A -> bool;
  eqf_leibniz : forall x y : A, reflect (x = y) (eqf x y);
}.
```

Advanced Coq C-4.5.1: Reflect

Reflect is an inductive type for relating propositions and booleans [27].

```
Inductive reflect (P : Prop) : bool -> Set :=
| ReflectT : P -> reflect P true
| ReflectF : (P -> False) -> reflect P false.
```

Function F-4.5.1: Normalization function for free group

```
Fixpoint normalize {A : Type} `{EqDec A}
  (x : NaiveFreeGroup A) : NaiveFreeGroup A :=
  match x with
  | []           => []
  | (b, v) :: x' =>
      match normalize x' with
      | []              => [(b, v)]
      | (b', v') :: x'' =>
          if andb (eqf v v') (xorb b b')
          then x''
          else (b, v) :: (b', v') :: x''
      end
  end.
```

4.5.2 The normalized representation

Based on the normalization process, we will define an inductive type where two of the same elements with opposite signs cannot be next to each other. In order to define this type, we first need to formalize the group in Coq.

Definition D-4.5.5: Type class of groups

```
Class Group (A : Type) := GroupDef {
  zero     : A;
  op       : A -> A -> A;
  inv      : A -> A;
  l_op_id  : forall x : A, op zero x = x;
  r_op_id  : forall x : A, op x zero = x;
  l_op_inv : forall x : A, op (inv x) x = zero;
  r_op_inv : forall x : A, op x (inv x) = zero;
  op_assoc : forall x y z : A, op (op x y) z = op x (op y z);
}.

Class EqGroup (A : Type) `{E : EqDec A} `{G : Group A} := {}.

Definition sub {A : Type} `{Group A} (x y : A) := op x (inv y).
```

We can make the following observations if the underlying type is a group. First, we notice that two elements are equal if and only if the difference between them is a neutral element. The second one is that we can define any list as a list of differences between elements. Similar to the normalization process, the list of differences starts with the last element, and then differences from the previous elements are added.

Definition D-4.5.6: Normalized representation of free group

```
Inductive NEQuotFreeGroup (A : Type) `{EqGroup A} :=
| Singl  : bool -> A -> NEQuotFreeGroup A
| Switch : forall x : A, Squash (x <> zero) ->
             NEQuotFreeGroup A -> NEQuotFreeGroup A
| Stay   : A -> NEQuotFreeGroup A -> NEQuotFreeGroup A.

Definition QuotFreeGroup (A: Type) `{EqGroup A}
  := option (NEQuotFreeGroup A).
```

This inductive type `NEQuotFreeGroup`, which contains non-empty normalized words (elements of a free group), is made of three constructors. The first one `Singl` is the constructor of a single element word. The second one `Switch` is the most interesting because it prevents the existence of two elements next to each other that differ only by a sign. It is used to add an element with the opposite sign. It is made of a difference between the added element and the head of the word and a proof that this difference is not the neutral element. This proof is squashed (see C-2.3.4), so we do not need to worry about its uniqueness. The third one `Stay` is used to add an element with the same sign as the head of the word. It is made of a difference between the added element and the head of the word. This construction cannot be used to define a type of free group with the neutral element since we know that the inverse of neutral elements is a neutral element from the group's laws. Therefore, to define potentially empty words, we need to use a second definition utilizing an option. `None` is the empty list constructor.

Example E-4.5.2: Exotic integers

The exotic integer representation (defined in D-4.3.1) is the special case of a free group over the single element type (`QuotFreeGroup unit`).

To apply this construction in practice, it is necessary to define conversion functions since the normalized representation is difficult for users to read and write. Therefore, the best approach is to define a word in naive representation and convert it to normalized representation.

Function F-4.5.2: Conversion to normalized representation

```
Fixpoint to_quot' {A : Type} `{EqGroup A} (b : bool) (v: A)
  (x : NaiveFreeGroup A) : NEQuotFreeGroup A :=
  match x with
  | []             => Singl b v
  | (b', v') :: x' =>
    if eqb b b'
    then Stay (sub v v') (to_quot' b' v' x')
    else match eqf_leibniz (sub v v') zero with
         | ReflectF _ p => Switch
             (sub v v') (squash p) (to_quot' b' v' x')
         | _            => (Singl b v) (* invalid *)
         end
end.
```

Function F-4.5.3: Conversion to normalized representation

```
Definition to_quot {A : Type} `{EqGroup A}
  (x : NaiveFreeGroup A) : QuotFreeGroup A :=
match x with
| []           => None
| (b, v) :: x' => Some (to_quot' b v x')
end.
```

In the definition of the function `to_quot'`, we need to prove that two consecutive elements are distinct when they differ by sign. Fortunately, using `reflect` in the decidable equality definition allows us to extract such proof easily. The `to_quot` function works correctly only for normalized words, so the result is undefined when mutually inverse elements appear next to each other. In order to create a conversion function for any word, we first need to normalize it, which means composing the conversion function with the normalization function F-4.5.1.

Function F-4.5.4: Conversion to naive representation

```
Fixpoint to_naive' {A : Type} `{EqGroup A}
  (x : NEQuotFreeGroup A) : bool * A * list (bool * A) :=
  match x with
  | Singl b v     => ((b, v), [])
  | Stay v x'     =>
      match to_naive' x' with
      | ((b, v'), y)  => ((b, op v v'), (b, v') :: y)
      end
  | Switch v _ x' =>
      match to_naive' x' with
      | ((b, v'), y)  => ((negb b, op v v'), (b, v') :: y)
      end
  end.
```

Function F-4.5.5: Conversion to naive representation

```
Definition to_naive {A : Type} `{EqGroup A}
  (x : QuotFreeGroup A) : NaiveFreeGroup A :=
  match x with
  | None    => []
  | Some x' => let (h, t) := to_naive' x' in h :: t
  end.
```

The inverse function `to_naive` is simpler. It relies on adding consecutive differences to compute the consecutive values. For the `Stay` constructor, the sign of the previous element is copied, while for the `Switch` constructor, the opposite sign is used. Since the `to_naive'` function deals with non-empty free groups, it returns a non-empty list of elements implemented as a pair consisting of the head and a potentially empty list.

4.5.3 Free groups as algebraic groups

As we know, elements of a free group form a group where the operator is the concatenation of words. The neutral element is the empty word, and the inverse of some word is the same word but reversed and with opposite signs, following the inverse law in groups:

$$(xy)^{-1} = y^{-1}x^{-1}$$

We can define the appropriate operations and demonstrate that they satisfy the group laws. These operations can be defined on the normalized representation. However, because the normalized representation is a list of differences, operations such as concatenation or reversing the order are difficult to define and inefficient. Therefore, we will define these operations on the naive representation of free groups, ensuring that the result for normalized arguments is also normalized.

Function F-4.5.6: Insertion

```
Definition naive_cons {A : Type} `{EqGroup A} (b : bool)
  (v : A) (x : NaiveFreeGroup A) : NaiveFreeGroup A :=
  match x with
  | []             => [(b, v)]
  | (b', v') :: t => if negb (eqf v v') || eqb b b'
                     then (b, v) :: (b', v') :: t
                     else t
  end.
```

Function F-4.5.7: Concatenation

```
Fixpoint naive_concat {A : Type} `{EqGroup A}
  (x y : NaiveFreeGroup A) : NaiveFreeGroup A :=
  match x with
  | []            => y
  | (b, v) :: x' => naive_cons b v (naive_concat x' y)
  end.
```

The concatenation of words `naive_concat` works similarly to the concatenation of lists, where elements from the first list are transferred to the second. However, in the case of normalized words, adding an element at the beginning of a word can remove its head if it is the same element but with the opposite sign.

Function F-4.5.8: Inversion

```
Fixpoint naive_apinv {A : Type} `{EqGroup A}
  (x y : NaiveFreeGroup A) : NaiveFreeGroup A :=
  match x with
  | []            => y
  | (b, v) :: x' => naive_apinv x' ((negb b, v) :: y)
  end.

Definition naive_inv {A : Type} `{EqGroup A}
  (x : NaiveFreeGroup A) : NaiveFreeGroup A := naive_apinv x [].
```

Similarly as in case of reversing regular lists, in order to define the inverse function for words (`naive_inv`) in linear time, we will use an ancillary function called naive_apinv. This function transfers elements from one list to another with their signs changed. The result is also normalized since we assume that the input is normalized. To define these operations for normalized representation, we need to compose functions defined above with conversion functions F-4.5.5 and F-4.5.3. In the appendix **FreeGroup.v**, we can find proof that the operations defined above satisfy the laws of groups. Moreover, we can also find there proof that there is a single normalized representation for each word.

4.5.4 Free groups as monads

Definition D-4.5.7: Monad

```
Class Monad (M : Type -> Type) := MondadDef {
  pure        : forall {A : Type}, A -> M A;
  bind        : forall {A B : Type}, M A -> (A -> M B) -> M B;
  left_bind   : forall {A B : Type} (f : A -> M B) (x : A),
                 bind (pure x) f = f x;
  right_bind : forall {A B : Type} (x : M A), bind x pure = x;
  comp_bind   : forall {A B C : Type} (f : B -> M C)
                 (g : A -> M B) (x : M A), bind (bind x g) f =
                 bind x (fun y => bind (g y) f);
}.
```

Every free group is also a monad and, therefore, a functor. However, for normalized representation, free group is a monad only for types with the group interface D-4.5.5. The pure function can be defined as the application of a singleton constructor.

Function F-4.5.9: Pure function

```
Definition naive_pure {A : Type} (x : A) : NaiveFreeGroup A
  := [(true, x)].
```

The bind function for free group is defined as:

$$\mathrm{bind}_f\ (a_1 \circ a_2^{-1} \circ \cdots \circ a_n) = f(a_1) \circ f(a_2)^{-1} \circ \cdots \circ f(a_n)$$

Function F-4.5.10: Bind function

```
Fixpoint naive_bind {A B : Type} `{EqGroup A} `{EqGroup B}
  (x : NaiveFreeGroup A) (f : A -> NaiveFreeGroup B)
  : NaiveFreeGroup B :=
  match x with
  | []               => []
  | (true , v) :: x' => naive_concat (f v) (naive_bind x' f)
  | (false, v) :: x' =>
      naive_concat (naive_inv (f v)) (naive_bind x' f)
  end.
```

The proof that those definitions satisfy monad laws can be found in the appendix **Extras/FreeGroupMonad.v**.

4.6 Finite multisets as lists of differences

In the previous chapter, we defined a multiset type using subtyping. The question arises if there is a definition of finite multisets that do not require dependant types since this concept is not available in many programming languages. For this purpose, we utilize a list of differences and characterize the underlying types for which it is possible to define such lists. The concept of storing differences and recovering values from them is known in computer science as differential backups [9]. However, using it for multiset representation has not been found in any paper read during writing this thesis.

4.6.1 An alternative normalization

We first define a normalization function for multisets of natural numbers to derive a normalized representation.

Function F-4.6.1: Minimal value in the list

```
Fixpoint min (h : nat) (l : list nat) : nat :=
  match l with
  | []         => h
  | (h' :: l') => if h' <=? h then min h' l' else min h l'
  end.
```

Function F-4.6.2: Subtract constant from every element of the list

```
Fixpoint sub_list (s : nat) (l : list nat) : list nat :=
  match l with
  | []        => []
  | (h :: l') => (h - s) :: sub_list s l'
  end.
```

Function F-4.6.3: Remove one zero from the list

```
Fixpoint rm_one_zero (l : list nat) : list nat :=
  match l with
  | []        => []
  | (h :: l') => if h =? O then l' else h :: rm_one_zero l'
  end.
```

Function F-4.6.4: Optimized selection sort

```
Fixpoint select_sort' (f : nat) (prev : nat) (l : list nat)
  : list nat :=
  match f with
  | O    => []
  | S f' =>
    match l with
    | []        => []
    | (h :: l') =>
        let m := min h l' in
        let p := prev + m in
        p :: (select_sort' f' p (rm_one_zero (sub_list m l)))
    end
end.

Definition select_sort (l : list nat) : list nat :=
  select_sort' (length l) O l.
```

For normalization, we use a modified selection sort algorithm for natural numbers. The modification optimizes the comparison operations by subtracting the found minimum from each member of the yet-to-be-sorted part of list. Comparing natural numbers in unary representation is linear with respect to the smaller number. As a result, the smaller the numbers on the list, the faster the process of finding the minimum becomes. Since the previously found minimum reduces the values on the sorted list, the list of the consecutive minimums this algorithm finds is a list of consecutive differences of the sorted input list.

4.6.2 The normalized representation

We can generalize this normalization process for all underlying types for which a unique symmetric difference between elements can be computed. For this, an underlying type needs to have a linear order defined. Otherwise, computing a unique minimum would be impossible.

Definition D-4.6.1: Type with computable unique diffrences

```
Class Diff (A : Type) `{E : EqDec A} := {
  D    : Type;
  diff : A -> A -> D;
  add  : A -> D -> A;

  diff_comm     : forall (x y : A), diff x y = diff y x;
  diff_def      : forall (x : A) (d : D), d = diff x (add x d);
  recover       : forall (x y : A),
    y = add x (diff x y) \/ x = add y (diff x y);
  diff_anti_sym : forall (x : A) (d : D),
    x = add (add x d) d -> d = diff x x;
  diff_trans    : forall (x y z : A),
    z = add (add x (diff x y)) (diff y z) -> z = add x (diff x z)
}.
```

In this paper, the definition above has been used. It implies the existence of a linear order without requiring it as a premise. However, it does require a type of symmetric difference – `D`, a function that computes such a difference for two elements – `diff`, and a function that adds the indicated difference to an element – `add`. Additionally, this function should meet five requirements. The first requirement enforces the symmetry of the difference through the commutative law of the difference operation. The second law defines the relationship between the symmetric difference operation and addition. The third one requires that at least one of the elements can be recovered from the symmetric difference. The fourth states that if we add a difference twice and the result does not change, that difference is neutral. The third and fourth laws imply the antisymmetry of the generated order. The fifth one requires our symmetric differences to be transitive, meaning that if we can recover a value by adding two differences with a common end, we can also recover it from the direct difference. We can use this definition to define a linear order: two elements are in a relation if and only if by adding their common difference to the first element we obtain the second element D-4.6.2. The proof of this fact can be found in the appendix **DifferentialLists.v**.

Definition D-4.6.2: Linear order induced by `Diff` type class

```
Global Program Instance Diff_is_LO (A: Type) `{D: Diff A}
  : LinearOrder A := {
  ord := fun x y => eqf y (add x (diff y x))
}.
```

Having the class of types with computable differences, we can define the type of multisets based on this definition. It is an option on the head of the list and the list of consecutive differences.

Definition D-4.6.3: Normalized representation of multisets

```
Definition DiffList (A : Type) `{Diff A} :=
  option (A * list D).
```

For this definition, we can define conversion functions.

Function F-4.6.5: Conversion to normalized representation

```
Fixpoint to_diff' {A : Type} `{Diff A} (p : A) (l : list A)
  : list D :=
  match l with
  | []       => []
  | (h :: t) => diff p h :: to_diff' h t
  end.
```

Function F-4.6.6: Conversion to normalized representation

```
Definition to_diff {A : Type} `{Diff A} (l : list A)
  : DiffList A :=
  match l with
  | []       => None
  | (h :: t) => Some (h, to_diff' h t)
  end.
```

The function `to_diff` computes consecutive differences, assuming that the input is sorted. In order to get a normalized representation for a multiset of any list, we need to compose it with a sort function F-3.2.3.

Function F-4.6.7: Conversion of list of differences to list

```
Fixpoint from_diff' {A : Type} `{Diff A} (x : A) (l : list D) :=
  match l with
  | []      => [x]
  | h :: l' => x :: from_diff' (add x h) l'
  end.
```

Function F-4.6.8: Conversion of list of differences to list

```
Definition from_diff {A : Type} `{Diff A} (x : DiffList A)
  : list A :=
  match x with
  | None        => []
  | Some (a, l) => (from_diff' a l)
  end.
```

The function `from_diff` computes the list of elements by adding all the differences. The proof that for every multiset, exactly one list of differences exists can be found in the appendix **DifferentialLists.v**.

4.6.3 Basic operations

Having the conversion functions defined, we can use all functions defined for lists as functions for multisets. That means our representation is, for example, a monad (for types that implement D-4.6.1). However, using conversion functions is computationally expensive. Therefore, we define an insertion operation natively on normalized representation.

Function F-4.6.9: Insertion

```
Fixpoint add_to_diff' {A : Type} `{Diff A} (x : A) (h : A)
  (l : list D) : list D :=
  match l with
  | []      => [diff h x]
  | d :: l' => if ord x (add h d)
               then diff h x :: diff x (add h d) :: l'
               else d :: add_to_diff' x (add h d) l'
  end.
```

Function F-4.6.10: Insertion

```
Definition add_to_diff {A : Type} `{Diff A} (x : A)
  (l : DiffList A) : DiffList A :=
  match l with
  | None         => Some (x, [])
  | Some (h, l) => if ord x h
                   then Some (x, diff x h :: l)
                   else Some (h, add_to_diff' x h l)
  end.
```

The function `add_to_diff` finds a position where the next value is greater than the adding element. In this position, it computes the difference between the previous value and the adding element and between the adding element and the next value. Next, it replaces the previous difference with two new ones. In function F-4.6.9, `ord` is the linear order defined by symmetric differences (as defined in D-4.6.2). We can prove that after applying such a function to a list, the new element is present, no element has been removed, and the length of the list has increased by one. The proofs of these facts can also be found in the appendix **DifferentialLists.v**.

Chapter 5

Function types as quotient types

In this chapter, we will explore how to define quotient-like types using functions. However, these representations often lead to more problems than solutions. Nonetheless, they are intriguing examples of innovative approaches to quotient types in Coq. These examples are sourced from Coq's set library [27].

5.1 Quotient-like functions

At the beginning of the thesis, we mentioned that it is impossible to define the type of sets and multisets for any arbitrary underlying type. It is true for inductive types, where the order of constructors matters. However, we can utilize the built-in function type. Of course, to reason about the equalities of functions in a sensible manner, we will need to assume the axiom of function extensionality. Introducing this axiom does not introduce any contradictions to the Coq system, so we can safely add it.

Definition D-5.1.1: Function extensionality

```
Definition FunExt := forall (A B : Type) (f g : A -> B),
    (forall x : A, f x = g x) -> f = g.
```

Without this axiom, two functions are equal if and only if they are intentionally equal, which means they perform computations in the same way. With this axiom, two functions are equal if and only if they give the same results. We can use this property to "forget" the order of elements since the order of computations does not matter as long as it does not change the result.

5.2 Sets and multisets

Knowing this concept, we can define the set as the set characteristic function that decides whether the element is in the set or not.

Definition D-5.2.1: Set as characteristic function

```
Definition set (A : Type) : Type := A -> bool.
```

In contrast to previous definitions of sets, this one lets us define sets with potentially infinitely many elements. We can use this representation to reason about any set with a computable characteristic function. The cost of such a general definition is that some basic functions on set are uncomputable for this representation.

Theorem T-5.2.1: Checking emptiness

The function $E : \mathrm{set}(X) \to \{0, 1\}$, which checks whether the set is empty, is uncomputable.

Proof P-5.2.1

Let E be a computable function. For any Turing machine M there exists set $T_M = \{t \in \mathbb{N} : \text{after } t \text{ steps machine } M \text{ is in HALT state}\}$. Every set T_M has a computable characteristic function since checking if the Turing machine halts after a specific number of steps is a computable problem. Checking if T_M is empty is equivalent to checking if the Turning machine ever halts. However, this is an undecidable problem. Therefore, function E is not computable.

↯

Multisets can be defined similarly by replacing codomain type from booleans to natural numbers.

Definition D-5.2.2: Multisets as characteristic function

```
Definition mset (A : Type) : Type := A -> nat.
```

We can express in this representation only multisets that have finitely many copies of each element of the underlying type. In order to generalize this representation, we need to replace natural numbers with co-natural numbers.

Advanced Coq C-5.2.1: Co-natural numbers

In Coq, *co-natural numbers* can be defined in the following way:

```
CoInductive conat : Type :=
| O' : conat
| S' : conat -> conat.
```

In contrast to an inductive definition of natural numbers, co-natural numbers are coinductive. That means we define the destructor instead of constructors. Destructor produces from a co-natural number either `O'` – zero or `S' n` – the successor of co-natural number n. Since we use a destructor, we can define a number that is a successor of itself.

```
CoFixpoint inf : conat := S' inf.
```

This number can be called infinity. We can also define any finite number.

```
Fixpoint toConat (n : nat) : conat :=
  match n with
  | O => O'
  | S n' => S' (toConat n')
  end.
```

5.3 Basic operations

As we showed in Theorem T-5.2.1, checking if the set is empty is an undecidable problem. Comparing two sets is an uncomputable operation since we can compare a set to an empty set to check if it is empty. Another missing operation is a mapping operation. We can check emptiness by mapping every element to the element of the unit type if a mapping is computable. Despite those problems, we can still define helpful functions for this set representation. Checking if some element is in a set is trivial since we represent a set as a characteristic function. We can easily define the filtering function.

Function F-5.3.1: Filtering

```
Definition set_filter {A : Type} (p : A -> bool)
  (s : set A) : set A :=
  fun x : A => if p x then s x else false.
```

This definition utilizes lazy evaluation. Therefore, potential infinity is not a problem. Similarly, we can define union, intersection, and complement of a set.

Function F-5.3.2: Union

```
Definition set_union {A : Type} (s s' : set A) : set A :=
  fun x : A => (s x) || (s' x).
```

Function F-5.3.3: Intersection

```
Definition set_intersection {A : Type} (s s' : set A) : set A :=
  fun x : A => (s x) && (s' x).
```

Function F-5.3.4: Complement

```
Definition set_complement {A : Type} (s : set A) : set A :=
  fun x : A => negb (s x).
```

Those operations can also be implemented similarly for multisets using sum and minimum operations. The complement operation is not defined for multisets. The drawback of these lazy computations is that with each subsequent intersection and union, the complexity of the set characteristic function increases. The operation of adding an element to the set can also be defined. However, it requires the underlying type with decidable equality.

Function F-5.3.5: Insertion

```
Definition set_add {A : Type} `{EqDec A} (a : A) (s : set A)
  : set A :=
  fun x : A => if eqf x a then true else s x.
```

Function F-5.3.6: Conversion to a set

```
Fixpoint list_to_set {A : Type} `{EqDec A} (l : list A)
  : set A :=
  match l with
  | []        => fun _ => false
  | (h :: l') => set_add h (list_to_set l')
  end.
```

Proofs of fundamental properties of those operations can be found in the appendix **FunctionalQuotient.v**.

Chapter 6

Quotient types in selected languages

In the previous chapters, we discussed how we can define quotient-like types in Coq. In this chapter, we will examine languages that have built-in support for quotient types. We will learn how they are implemented and whether similar mechanisms can be utilized in Coq.

6.1 Lean

Lean is a proof assistant under development since 2013 [4]. It is an open-source tool for verifing program reliability and mathematical theorems, similar to Coq. Due to the similarity of both languages, we will translate Lean's axioms into the Coq language.

Advanced Coq C-6.1.1: Differences compared to Coq

Unlike Coq, Lean does not require constructivism of theorems [4]. It means we are dealing with a system closer to mathematical proofs than programming. Moreover, in Lean, we have definitional irrelevance of propositions. The formal system is based on three axioms:

- Propositional Extensionality

  ```
  Axiom prop_ext: forall P Q : Prop, (P <-> Q) <-> (P = Q)
  ```

- Axiom of choice

  ```
  Inductive NEmpty (A : Type) : Prop := intro : A -> NEmpty A.
  Axiom choice: forall A: Type, NEmpty A -> A.
  ```

- Quotient types

  ```
  Axiom Quot : forall {A : Type}, (A -> A -> Prop) -> Type.
  ```

One of the main differences between Lean and Coq is the use of classical logic in Lean. Therefore, one might assume that the law of excluded middle should be one of the axioms in this system. However, the three axioms mentioned above are sufficient to derive the law of excluded middle and the extensionality of functions. Extensionality of functions can be derived from quotients, and the law of excluded middle can be derived from the axiom of choice as proposed by Diaconescu [21]. These proofs can also be found in the Lean standard library under the names `em` and `funext`.

6.1.1 Axioms of quotient types

As we saw in C-6.1.1, quotient types are integral with Lean language. Let us take a closer look at the axioms that define them.

Definition D-6.1.1: Axioms of Lean's quotient types

The first axiom, named in Lean as `Quot`, states that the quotient type exists for every relation and underlying type.

```
Axiom Quot : forall {A : Type}, (A -> A -> Prop) -> Type.
```

The second axiom, named in Lean as `Quot.mk`, states that there exists an embedding function that, out of every element of the underlying type, makes an element of quotient type.

```
Axiom Quot_mk : forall {A : Type} (r : A -> A -> Prop),
  A -> Quot r.
```

The third axiom, named in Lean as `Quot.ind`, provides us with induction law for quotient types. It states that every element of the underlying type is made of an element of the underlying type.

```
Axiom Quot_ind :
  forall (A : Type) (r : A -> A -> Prop) (P : Quot r -> Prop),
  (forall a : A, P (Quot_mk r a)) -> forall q: Quot r, P q.
```

The fourth axiom, named in Lean as `Quot.lift`, describes how functions on the underlying type can be applied to the quotient type. It requires that such function respect quotient-defining relation. In Lean, this axiom comes with a rule that application on quotient type works the same way as on the underlying type. In Coq, this rule needs to be added as an additional axiom.

```
Axiom Quot_lift :
  forall (A : Type) (r : A -> A -> Prop) (B : Type) (f : A -> B),
  (forall a b : A, r a b -> f a = f b) -> Quot r -> B.

Axiom Quot_lift' : forall {A : Type} {r : A -> A -> Prop}
  {B : Type} (f : A -> B) (P : forall a b: A, r a b -> f a = f b)
  (x : A), f x = Quot_lift f P (Quot_mk r x).
```

The fifth axiom, named in Lean as `Quot.sound`, states that `Quot` is indeed a quotient type – if two elements are in the quotient-defining relation, they share the same element in the quotient type.

```
Axiom Quot_sound :
  forall (A : Type) (r : A -> A -> Prop) (a b : A),
  r a b -> Quot_mk r a = Quot_mk r b.
```

As we see in the Definition D-6.1.1, axioms of Lean's quotient types can be expressed in Coq. Therefore, we can use them to add quotient type to the Coq language.

6.2 Agda

Agda is a programming language created to support dependent types [35]. It originated as an extension of Martin-Löf's type theory [33]. Due to these features, Agda can be used as a proof assistant. However, unlike languages such as Coq or Lean, Agda does not have a tactic language, which significantly complicates its use as such. On the other hand, handling dependent types in Agda is at a much higher level than in Coq. Agda can often infer that a particular case is impossible, making the definition of dependent functions much more accessible than in Coq.

6.2.1 Higher inductive types

In standard Agda, there is no built-in mechanism for defining quotient types. Therefore, in this section, we focus on *Cubical* extension that introduces known from homotopic type theory higher inductive types.

Definition D-6.2.1: Higher inductive types

Higher inductive types (HITs) are a generalization of inductive types that allow constructors to produce not just elements of the defined type but also elements of its iterated identity types [28].

Example E-6.2.1: Circle

The *circle* is a higher inductive type made out of single element `base` and extra identity proof `loop` [28]. In homotopic interpretation, this is a point with a loop. Therefore, it is called a circle.

```
data S : Set where
  base : S
  loop : base ≡ base
```

Example E-6.2.2: Interval

The *interval* is a higher inductive type made out of two elements `left` and `right`, that are connected by single identity proof `segment` [28].

```
data interval : Set where
  left    : interval
  right   : interval
  segment : left ≡ right
```

Example E-6.2.3: Suspension

The *suspension* is a higher inductive type made out of two elements `north` and `south`, and paths for every element of underlying types [28].

```
data susp (A : Set) : Set where
  north : susp A
  south : susp A
  merid : A -> north ≡ south.
```

6.2.2 Quotient types utilizing HITs

Higher inductive types solve the fundamental problem of inductive types: the inability to glue equivalent elements. Using inductive types, we can only add new elements via constructors. If two different constructions represent the same value, we need to use some quotient type to glue equivalent elements. Using higher inductive types, we can combine those two steps and add constructors for additional equalities (paths) for the inductive type that is being defined.

Example E-6.2.4: Multiset

```
data MSet (A : Set) : Set where
  []    : MSet A
  _::_ : A -> MSet A -> MSet A
  swap : (x y : A) (z : MSet A) -> x :: y :: z ≡ y :: x :: z
```

The higher inductive types' downside is that they complicate defining functions since we also need to handle constructors for identity proofs. Otherwise, we could define functions that do not respect identities (paths).

Example E-6.2.5: Union

```
_ ∪ _ : (xs ys : MSet A) -> MSet A
xs ∪ [] = xs
xs ∪ (y :: ys) = y :: (xs ∪ ys)
xs ∪ (swap y y' ys i) = swap y y' (xs ∪ ys) i
```

As we see in the Example E-6.2.5, to define a multiset union, we also need to define a case when a multiset is on the path that swaps its heads. For both ends of this path, the result must be the same. Therefore, defining the head function for multisets is impossible.

Advanced Coq C-6.2.1: HIT in Coq

In Coq, we can also define higher inductive types. However, they require using private inductive types, proposed by Yves Bertot [11]. Private inductive types let us define the inductive types and hide their definitions. They turn off pattern matching on those inductive types, forcing users to use only exposed functions. Coq's library for homotopy type theory, including higher inductive types [8], was created using private inductive types.

Chapter 7

Summary

Quotient types hold significant importance in the field of mathematics formalization and theorem proving. Our thesis highlights that the Coq programming language lacks a built-in method to define quotient types. Hence, we must utilize other constructions with some advantageous properties of quotient types. Our thesis mainly focused on quotient-like types, where equality is equivalent to a quotient-defining equivalence relation. We extensively examined different methods of defining such types using normalization functions.

7.1 Normalization functions

The normalization function is a helpful tool for defining quotient-like types. Most quotient types that are frequently used have computable normalization functions that can be utilized for the purpose of defining quotient-like types. However, certain quotient types, such as general unordered pairs or potentially infinite sets, do not have a normalization function. Nevertheless, as demonstrated in the fifth chapter, defining quotient-like types in Coq is possible in such cases, but it usually comes with many disadvantages. For instance, in the case of sets determining whether the set is empty is an undecidable problem. The class of quotient types with computable normalization function is expansive enough to assess various approaches to constructing quotient-like types.

7.2 Subtypes

Subtypes are helpful in the case of defining types with a subset of elements of the underlying type. They are a dual construction to quotient types and they are already available in Coq. They can be easily applied to restrict an underlying type to a subset of normalized elements, assuming we have a normalization function or predicate.

Defining a subtype with only normalized elements is straightforward. However, proving only a single element is in the subtype for every equivalence class is difficult due to the fact that `Prop` sort and proof irrelevance axiom are independent. This makes proving the uniqueness of representations in the general case impossible. One possible solution is to add the proof irrelevance axiom to the system. However, there exist other solutions that do not require additional axioms. For instance, we can squash (as in C-2.3.4) the normalization predicate to ensure that it is inhabited by, at most, a single witness. Nevertheless, this makes proving other subtype properties much more complicated if not impossible. The final solution is to prove that the underlying type of the subtype has unique identity proofs. In such cases, we can define a quotient-like construction and proof that it satisfies the necessary properties for the uniqueness of representations (as in D-2.3.1).

Defining quotient-like constructions using subtypes is easy when types have decidable equality and normalization functions. However, defining operations for this construction can be complicated. One solution is to always normalize the result after applying any operation. Another solution is to prove that the result of the operation is normalized if the input is normalized. This approach is recommended if we refrain from using squashed proofs of normalization for subtype definition. It lets us use properties of functions proven on the underlying type without extra effort.

7.3 Traces of normalization functions

In this thesis, a different method for defining quotient-like types is presented. It involves creating tailor-made inductive types using the traces of the normalization process. When constructing these types, it is easy to prove the uniqueness of their representations.

A challenge in this approach is creating a customized inductive type for a quotient type. In chapter four, we give examples of such constructions based on traces of the normalization function. However, generalizing the process of constructing types by examining the normalization function is challenging. Only some steps of the normalization process are relevant for creating a quotient-like type, as most steps depend on which unnormalized form was used.

An example of this problem is the normalization process of integers. The final step alone creates a quotient-like type. The inspiration for this thesis was an inductive type of rational numbers based on traces of the Euclidean algorithm. This approach of using traces of normalization function solved a nontrivial problem. The same can be said for traces of normalization of a free monoid. However, it is uncertain whether there is a connection between the traces of the normalization function and some quotient-like types or whether this was a mere coincidence for those examples.

It is unlikely to find a connection between modified sorting functions and the type of lists of differences. This thesis concludes that there is no general way of deriving a normalized representation based on normalization. However, more research is needed to prove this due to the difficulty of establishing a lack of connection.

One of the challenges with the tailor-made quotient-like types is defining basic operations. Although it was easy to define operations for some types, such as free monoids and integers, it proved difficult for most types. As a result, for defining operations, we had to use a naive representation as an intermediary form. To use operations defined for naive representations, we must demonstrate that they maintain the predicate of normalization.

7.4 Related and further works

This thesis covers numerous concepts related to quotient types in Coq. However, it only includes some constructions. For learning the state-of-the-art practical approach to this problem, we recommend reading [17]. This paper explains how to combine the advantages of setoids and quotient-like types. Another essential paper on this topic is [19], which, like this thesis, emphasizes the importance of normalization functions in defining quotient-like types. For those interested in the normalization approach, we also suggest reading [2], which discusses the limitations and possibilities of defining quotient types.

The main inspiration for writing this thesis was [10]. This paper describes using traces of the Euclidean algorithm to define the quotient-like type for rational numbers. A similar approach is described in [40] for Hereditarily Finite Sets. However, we have come to the conclusion that trying to find a universal way of using traces of the normalization function as quotient-like types is a dead end. It is important to note that we can only discover something new by having a small amount of skepticism regarding any information presented to us. Therefore, we encourage readers to verify this conclusion on their own.

We have a couple of recommendations for people interested in quotient types in type theory, regardless of whether they are applicable in Coq or not. Firstly, we suggest reading [5]. It is a fantastic introduction to type theory. Moreover, it describes how to define congruence types, a similar concept to quotient types. Next, we recommend reading [16]. This paper describes which isomorphisms can and cannot be defined for quotient types. We also suggest reading paper [34], which is about quotient types in the NuPRL language. It excellently describes the difference between intentional and extensional approach to quotient types.

7.5 Conclusions

Based on our analysis of various methods for defining quotient-like types in Coq, we have found that their practical usage is limited. Subtypes are more suitable for computational purposes. However, we need to prove the normalization predicate's uniqueness to use them as quotient-like types. Even after that, we need to ensure that the operation preserves the predicate of normalization. This requirement is almost as challenging as proving that the operation respects an equivalence relation. Therefore, sentoids are a better approach in most everyday applications since they can be defined for every equivalence relation without the hardship of defining the normalization function or predicate. However, the downside is that we cannot use equality relations in our prepositions. If we want to use equality relation, a better solution to quotient-ike necessarily types is to incorporate them via axioms, as demonstrated in the sixth chapter, or to use a private inductive type, as in the homotopy type theory (HoTT) library for Coq. Therefore, defining quotient-like types remains an interesting concept, but primarily for academic purposes.

Bibliography

[1] A. Abel and J. Chapman. Normalization by evaluation in the delay monad: A case study for coinduction via copatterns and sized types. In P. B. Levy and N. Krishnaswami, editors, *Proceedings 5th Workshop on Mathematically Structured Functional Programming, MSFP@ETAPS 2014, Grenoble, France, 12 April 2014*, volume 153 of *EPTCS*, pages 51–67, 2014.

[2] T. Altenkirch, T. Anberrée, and N. Li. Definable quotients in type theory. `http://www.cs.nott.ac.uk/~psztxa/publ/defquotients.pdf`, 2011.

[3] T. Altenkirch, S. Boulier, A. Kaposi, and N. Tabareau. Setoid type theory—a syntactic translation. In G. Hutton, editor, *Mathematics of Program Construction*, pages 155–196, Cham, 2019. Springer International Publishing.

[4] J. Avigad, L. de Moura, S. Kong, and S. Ullrich. Theorem proving in lean 4. `https://leanprover.github.io/theorem_proving_in_lean4/axioms_and_computation.html`, 2022. Accessed: 12-08-2023.

[5] R. Backhouse, P. Chisholm, G. Malcolm, and E. Saaman. Do-it-yourself type theory. *Formal Aspects of Computing*, 1(1):19–4, 1989.

[6] G. Barthe. The relevance of proof-irrelevance. In K. G. Larsen, S. Skyum, and G. Winskel, editors, *Automata, Languages and Programming*, pages 755–768, Berlin, Heidelberg, 1998. Springer Berlin Heidelberg.

[7] G. Barthe, V. Capretta, and O. Pons. Setoids in type theory. *Journal of Functional Programming*, 13(2):261–293, 2003.

[8] A. Bauer, J. Gross, P. L. Lumsdaine, M. Shulman, M. Sozeau, and B. Spitters. The hott library: A formalization of homotopy type theory in coq. *CoRR*, abs/1610.04591, 2016.

[9] B. Beard. *Beginning Backup and Restore for SQL Server.* Apres, 01 2018.

[10] Y. Bertot. A simple canonical representation of rational numbers. *Electronic Notes in Theoretical Computer Science*, 85(7):1–16, 2003. Mathematics, Logic and Computation (Satellite Event of ICALP 2003).

[11] Y. Bertot. Private inductive types proposing a language extension. `https://coq.inria.fr/files/coq5_submission_3.pdf`, 2013.

[12] Y. Bertot and P. Castran. *Interactive Theorem Proving and Program Development: Coq'Art The Calculus of Inductive Constructions.* Springer Publishing Company, Incorporated, 1st edition, 2010.

[13] S. O. Biha. Finite groups representation theory with coq. In J. Carette, L. Dixon, C. S. Coen, and S. M. Watt, editors, *Intelligent Computer Mathematics, 16th Symposium, Calculemus 2009, 8th International Conference, MKM 2009, Held as Part of CICM 2009, Grand Bend, Canada, July 6-12, 2009. Proceedings*, volume 5625 of *Lecture Notes in Computer Science*, pages 438–452. Springer, 2009.

[14] E. Bishop. *Foundations of Constructive Analysis.* New York, NY, USA: Mcgraw-Hill, 1967.

[15] J. Boyarsky and S. Selikoff. *OCA Oracle Certified Associate Java SE 8 Programmer I study guide.* John Wiley and Sons, 2015.

[16] L. Chicli, L. Pottier, and C. Simpson. Mathematical quotients and quotient types in coq. In H. Geuvers and F. Wiedijk, editors, *Types for Proofs and Programs*, pages 95–107, Berlin, Heidelberg, 2003. Springer Berlin Heidelberg.

[17] C. Cohen. Pragmatic quotient types in coq. In S. Blazy, C. Paulin-Mohring, and D. Pichardie, editors, *Interactive Theorem Proving*, pages 213–228, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg.

[18] T. Coquand. An analysis of girard's paradox. In *Proceedings of the First Annual IEEE Symposium on Logic in Computer Science (LICS 1986)*, pages 227–236. IEEE Computer Society Press, June 1986.

[19] P. Courtieu. Normalized types. In L. Fribourg, editor, *Computer Science Logic*, pages 554–569, Berlin, Heidelberg, 2001. Springer Berlin Heidelberg.

[20] M. Davis. *The Undecidable: Basic Papers on Undecidable Propositions, Unsolvable Problems and Computable Functions.* Dover books on mathematics. Dover Publications, 2004.

[21] R. Diaconescu. Axiom of choice and complementation. *Proceedings of the American Mathematical Society*, 51(1):176–178, 1975.

[22] D. S. Dummit and R. M. Foote. *Abstract algebra.* Wiley, New York, 3rd ed edition, 2004.

[23] G. Gilbert, J. Cockx, M. Sozeau, and N. Tabareau. Definitional proof-irrelevance without k. *Proc. ACM Program. Lang.*, 3(POPL), jan 2019.

[24] M. Hedberg. A coherence theorem for martin-löf's type theory. *Journal of Functional Programming*, 8(4):413–436, 1998.

[25] M. Hofmann. *Extensional concepts in intensional type theory.* PhD thesis, University of Edinburgh, 1995.

[26] W. A. Howard. The formulae-as-types notion of construction. In H. Curry, H. B., S. J. Roger, and P. Jonathan, editors, *To H. B. Curry: Essays on Combinatory Logic, Lambda Calculus, and Formalism.* Academic Press, 1980.

[27] Inria, CNRS and contributors. Calculus of inductive constructions. `https://coq.inria.fr/refman/index.html`. Accessed: 2023-07-24.

[28] N. Institute for Advanced Study (Princeton and U. F. Program. *Homotopy Type Theory: Univalent Foundations of Mathematics.* Univalent Foundations Program, 2013.

[29] D. E. Knuth. *The Art of Computer Programming, Vol. 1: Fundamental Algorithms.* Addison-Wesley, Reading, Mass., third edition, 1997.

[30] N. Kraus, M. Escardó, T. Coquand, and T. Altenkirch. Generalizations of hedberg's theorem. In M. Hasegawa, editor, *Typed Lambda Calculi and Applications*, pages 173–188, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg.

[31] M. Lipovača. Learn you a haskell for great good! http://learnyouahaskell.com/, 2010.

[32] S. MacLane. *Categories for the Working Mathematician.* Springer-Verlag, New York, 1971. Graduate Texts in Mathematics, Vol. 5.

[33] P. Martin-Löf. An intuitionistic theory of types: Predicative part. In H. Rose and J. Shepherdson, editors, *Logic Colloquium '73*, volume 80 of *Studies in Logic and the Foundations of Mathematics*, pages 73–118. Elsevier, 1975.

[34] A. Nogin. Quotient types: A modular approach. In V. A. Carreño, C. A. Muñoz, and S. Tahar, editors, *Theorem Proving in Higher Order Logics*, pages 263–280, Berlin, Heidelberg, 2002. Springer Berlin Heidelberg.

[35] U. Norell, A. Abel, N. A. Danielsson, M. Takeyama, and C. Coquand. Agda documentation. `https://agda.readthedocs.io/en/v2.6.0.1/getting-started/what-is-agda.html`, 2021. Accessed: 12-08-2023.

[36] J. Roitman. *Introduction to Modern Set Theory.* A Wiley-interscience publication. Wiley, 1990.

[37] W. Rudin. *Principles of mathematical analysis / Walter Rudin.* McGraw-Hill New York, 3d ed. edition, 1976.

[38] D. Schepler. Freegroups coq contribution. `https://github.com/coq-contribs/free-groups`. Accessed: 27-07-2023.

[39] D. Singh, I. Adeku Musa, Y. Tella, and J. Singh. An overview of the applications of multisets. *Novi Sad J. Math*, 37:73–92, 01 2007.

[40] G. Smolka and K. Stark. Hereditarily finite sets in constructive type theory. In J. C. Blanchette and S. Merz, editors, *Interactive Theorem Proving*, pages 374–390, Cham, 2016. Springer International Publishing.

[41] T. Streicher. Investigations into intensional type theory. *Habilitiation Thesis, Ludwig Maximilian Universität*, 1993.

[42] The Idris Community. Idris 2 documentation. `https://idris2.readthedocs.io/en/latest/tutorial/index.html`, 2021. Accessed: 15-08-2023.

[43] B. Werner. *Une Théorie des Constructions Inductives.* Theses, Université Paris-Diderot - Paris VII, May 1994.