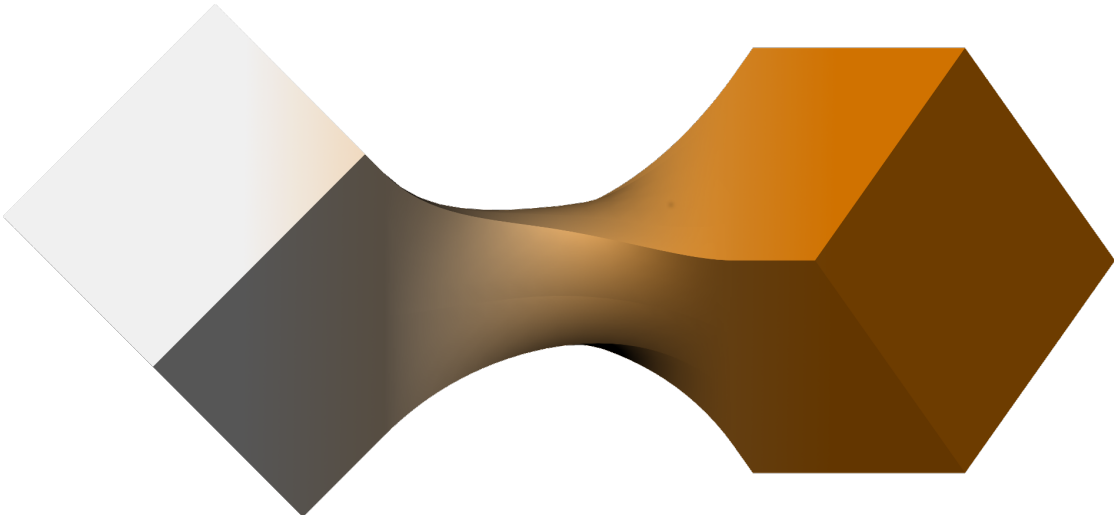
CHALMERS
UNIVERSITY OF TECHNOLOGY

UNIVERSITY OF GOTHENBURG

Univalent Categories

A formalization of category theory in Cubical Agda



Frederik Hanghøj Iversen

Master's thesis in Computer Science

Master's thesis 2018

Univalent Categories

A formalization of category theory in Cubical Agda

Frederik Hanghøj Iversen

CHALMERS
UNIVERSITY OF TECHNOLOGY

Department of Computer Science and Engineering
Programming Logic Group
Chalmers University of Technology and University of Gothenburg
Gothenburg, Sweden 2018

Univalent Categories
A formalization of category theory in Cubical Agda

Author:
Frederik Hanghøj Iversen
<hanghj@student.chalmers.se>

Supervisor:
Thierry Coquand
<coquand@chalmers.se>
Chalmers University of Technology and University of Gothenburg

Co-supervisor:
Andrea Vezzosi
<vezzosi@chalmers.se>
Chalmers University of Technology and University of Gothenburg

Examiner:
Andreas Abel
<abela@chalmers.se>
Chalmers University of Technology and University of Gothenburg

Master's Thesis 2018
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg
SE-412 96 Gothenburg
Telephone +46 31 772 1000
Gothenburg, Sweden 2018

Abstract

The usual notion of propositional equality in intensional type-theory is restrictive. For instance it does not admit functional extensionality nor univalence. This poses a severe limitation on both what is *provable* and the *re-usability* of proofs. Recent developments have, however, resulted in cubical type theory, which permits a constructive proof of univalence. The programming language Agda has been extended with capabilities for working in such a cubical setting. This thesis will explore the usefulness of this extension in the context of category theory.

The thesis will motivate the need for univalence and explain why propositional equality in cubical Agda is more expressive than in standard Agda. Alternative approaches to Cubical Agda will be presented and their pros and cons will be explained. As an example of the application of univalence, two formulations of monads will be presented: Namely monads in the monoidal form and monads in the Kleisli form. Using univalence, it will be shown how these are equal.

Finally the thesis will explain the challenges that a developer will face when working with cubical Agda and give some techniques to overcome these difficulties. It will suggest how further work can help alleviate some of these challenges.

Acknowledgements

I would like to thank my supervisor Thierry Coquand for giving me a chance to work on this interesting topic. I would also like to thank Andrea Vezzosi for some very long and very insightful meetings during the project. It is fascinating and almost uncanny how quickly Andrea can conjure up various proofs. I also want to recognize the support of Knud Højgaards Fond who graciously sponsored me with a 20.000 DKK scholarship which helped toward sponsoring the two years I have spent studying abroad. I would also like to give a warm thanks to my fellow students Pierre Kraft and Nachiappan Valliappan who have made the time spent working on the thesis way more enjoyable. Lastly I would like to give a special thanks to Valentina Méndez who have been a great moral support throughout the whole process.

Contents

Chapter 1

Introduction

This thesis is a case study in the application of cubical Agda to the formalization of category theory. At the center of this is the notion of *equality*. There are two pervasive notions of equality in type theory: *judgmental equality* and *propositional equality*. Judgmental equality is a property of the type system. Propositional equality on the other hand is usually defined *within* the system. When introducing definitions this report will use the symbol $\triangleq$. Judgmental equalities will be denoted with $=$ and for propositional equalities the notation $\equiv$ is used.

The rules of judgmental equality are related with β- and η-reduction, which gives a notion of computation in a given type theory. There are some properties that one usually want judgmental equality to satisfy. It must be *sound*, enjoy *canonicity* and be a *congruence relation*. Soundness means that things judged to be equal are equal with respects to the *model* of the theory or the *meta theory*. It must be a congruence relation, because otherwise the relation certainly does not adhere to our notion of equality. E.g. One would be able to conclude things like: $x \equiv y \to f\ x \not\equiv f\ y$. Canonicity means that any well typed term evaluates to a *canonical* form. For example, for a closed term $e : \mathbb{N}$, it will be the case that e reduces to n applications of *suc* to 0 for some n; i.e. $e = suc^n\ 0$. Without canonicity terms in the language can get "stuck", meaning that they do not reduce to a canonical form.

For a system to work as a programming languages it is necessary for judgmental equality to be *decidable*. Being decidable simply means that that an algorithm exists to decide whether two terms are equal. For any practical implementation, the decidability must also be effectively computable.

For propositional equality the decidability requirement is relaxed. It is not in general possible to decide the correctness of logical propositions (cf. Hilbert's *entscheidigungsproblem*).

There are two flavors of type-theory. *Intensional*- and *extensional*- type theory (ITT and ETT respectively). Identity types in extensional type theory are required to be *propositions*. That is, a type with at most one inhabitant. In extensional type theory the principle of reflection

$$a \equiv b \to a = b$$

is enough to make type checking undecidable. This report focuses on Agda, which at a glance can be thought of as a version of intensional type theory. Pattern-matching in regular Agda lets one prove *Uniqueness of Identity Proofs* (UIP). UIP states that any two identity proofs are propositionally identical.

The usual notion of propositional equality in ITT is quite restrictive. In the next section a few motivating examples will be presented that highlight. There exist techniques to circumvent these problems, as we shall see. This thesis will explore an extension to Agda that redefines the notion of propositional equality and as such is an alternative to these other techniques. The extension is called cubical Agda. Cubical Agda drops UIP, as it does not permit *functional extensionality* nor *univalence*. What makes cubical Agda particularly interesting is that it gives a *constructive* interpretation of univalence. What all this means will be elaborated in the following sections.

1.1 Motivating examples

In the following two sections I present two examples that illustrate some limitations inherent in ITT and, by extension, Agda.

1.1.1 Functional extensionality

Consider the functions:

$$\begin{aligned} \mathit{zeroLeft} &\triangleq \lambda\ (n : \mathbb{N}) \to (0 + n : \mathbb{N}) \\ \mathit{zeroRight} &\triangleq \lambda\ (n : \mathbb{N}) \to (n + 0 : \mathbb{N}) \end{aligned}$$

The term $n + 0$ is *definitionally* equal to n, which we write as $n + 0 = n$. This is also called *judgmental equality*. We call it definitional equality because the *equality* arises from the *definition* of $+$, which is:

$$\begin{aligned} + &: \mathbb{N} \to \mathbb{N} \to \mathbb{N} \\ n + 0 &\triangleq n \\ n + (\mathit{suc}\ m) &\triangleq \mathit{suc}\ (n + m) \end{aligned}$$

Note that $0 + n$ is *not* definitionally equal to n. This is because $0 + n$ is in normal form. I.e. there is no rule for $+$ whose left hand side matches this expression. We do, however, have that they are *propositionally* equal, which we write as $n \equiv n + 0$. Propositional equality means that there is a proof that exhibits this relation. We can do induction over n to prove this:

$$\begin{aligned} \mathit{zrn}\ &: \forall n \to n \equiv \mathit{zeroRight}\ n \\ \mathit{zrn}\ \mathit{zero} &\triangleq \mathit{refl} \\ \mathit{zrn}\ (\mathit{suc}\ n) &\triangleq \mathit{cong}\ \mathit{suc}\ (\mathit{zrn}\ n) \end{aligned} \tag{1.1.1}$$

This show that zero is a right neutral element (hence the name *zrn*). Since equality is a transitive relation we have that $\forall n \to \mathit{zeroLeft}\ n \equiv \mathit{zeroRight}\ n$. Unfortunately we don't have $\mathit{zeroLeft} \equiv \mathit{zeroRight}$. There is no way to construct a proof asserting the obvious equivalence of *zeroLeft* and *zeroRight*. Actually showing this is outside the scope of this text. It would essentially involve giving a model for our type theory that validates all our axioms but where $\mathit{zeroLeft} \equiv \mathit{zeroRight}$ is not true. We cannot show that they are equal even though we can prove them equal for all points. This is exactly the notion of equality that we are interested in for functions: Functions are considered equal when they are equal for all inputs. This is called *pointwise equality* where *points* of a function refer to its arguments.

1.1.2 Equality of isomorphic types

Let $\top$ denote the unit type – a type with a single constructor. In the propositions as types interpretation of type theory $\top$ is the proposition that is always true. The type $A \times \top$ and A has an element for each $a : A$. So in a sense they have the same shape (Greek; *isomorphic*). The second element of the pair does not add any "interesting information". It can be useful to identify such types. In fact it is quite commonplace in mathematics. Say we look at a set $\{x \mid \phi\ x \wedge \psi\ x\}$ and somehow conclude that $\psi\ x \equiv \top$ for all x. A mathematician would immediately conclude $\{x \mid \phi\ x \wedge \psi\ x\} \equiv \{x \mid \phi\ x\}$ without thinking twice. Unfortunately such an identification can not be performed in ITT.

More specifically what we are interested in is a way of identifying *equivalent* types. I will return to the definition of equivalence later in section §3.2, but for now it is sufficient to think of an equivalence as a one-to-one correspondence. We write $A \simeq B$ to assert that A and B are equivalent types. The principle of univalence says that:

$$univalence : (A \simeq B) \simeq (A \equiv B)$$

In particular this allows us to construct an equality from an equivalence

$$ua : (A \simeq B) \to (A \equiv B)$$

and vice versa.

1.2 Formalizing Category Theory

The above examples serve to illustrate a limitation of ITT. One case where these limitations are particularly prohibitive is in the study of Category Theory. At a glance category theory can be described as "the mathematical study of (abstract) algebras of functions" ([1]). By that token functional extensionality is particularly useful for formulating Category Theory. In Category theory it is also commonplace to identify isomorphic structures. Univalence gives us a way to make this notion precise. In fact we can formulate this requirement within our formulation of categories by requiring the *categories* themselves to be univalent as we shall see in section §3.3.

1.3 Context

The idea of formalizing Category Theory in proof assistants is not new. There are a multitude of these available online. Notably:

- A formalization in Agda using the setoid approach: `https://github.com/copumpkin/categories`
- A formalization in Agda with univalence and functional extensionality as postulates: `https://github.com/pcapriotti/agda-categories`
- A formalization in Coq in the homotopic setting: `https://github.com/HoTT/HoTT/tree/master/theories/Categories`

- A formalization in *CubicalTT* – a language designed for cubical type theory. Formalizes many different things, but only a few concepts from category theory: `https://github.com/mortberg/cubicaltt`

The contribution of this thesis is to explore how working in a cubical setting will make it possible to prove more things, to reuse proofs and to compare some aspects of this formalization with the existing ones.

There are alternative approaches to working in a cubical setting where one can still have univalence and functional extensionality. One option is to postulate these as axioms. This approach, however, has other shortcomings, e.g. you lose *canonicity* ([4, p. 3]).

Another approach is to use the *setoid interpretation* of type theory ([3, 4]). With this approach one works with *extensional sets* $(X, \sim)$. That is a type $X : \mathcal{U}$ and an equivalence relation $\sim : X \to X \to \mathcal{U}$ on that type. Under the setoid interpretation the equivalence relation serve as a sort of "local" propositional equality. Since the developer gets to pick this relation, it is not a priori a congruence relation. It must be manually verified by the developer. Furthermore, functions between different setoids must be shown to be setoid homomorphism, that is; they preserve the relation.

This approach has other drawbacks: It does not satisfy all propositional equalities of type theory a priori. That is, the developer must manually show that e.g. the relation is a congruence. Equational proofs $a \sim_X b$ are in some sense 'local' to the extensional set $(X, \sim)$. To e.g. prove that $x \sim y \to f\ x \sim f\ y$ for some function $f : A \to B$ between two extensional sets A and B it must be shown that f is a groupoid homomorphism. This makes it very cumbersome to work with in practice ([4, p. 4]).

1.4 Conventions

In the remainder of this thesis I will use the term *Type* to describe – well – types; thereby departing from the notation in Agda where the keyword `Set` refers to types. *Set*, on the other hand, shall refer to the homotopical notion of a set. I will also leave all universe levels implicit. This of course does not mean that a statement such as $\mathcal{U} : \mathcal{U}$ means that we have type-in-type but rather that the arguments to the universes are implicit.

I use the term *arrow* to refer to morphisms in a category, whereas the terms *morphism*, *map* or *function* shall be reserved for talking about type theoretic functions; i.e. functions in Agda.

As already noted $\triangleq$ will be used for introducing definitions $=$ will be used to for judgmental equality and $\equiv$ will be used for propositional equality.

All this is summarized in the following table:

Name	Agda	Notation
Type	`Set`	$\mathcal{U}$
Set	`Σ Set IsSet`	*Set*
Function, morphism, map	`A → B`	$A \to B$
Dependent- ditto	`(a : A) → B`	$\prod_{a:A} B$
Arrow	`Arrow A B`	*Arrow A B*
Object	`C.Object`	$\mathbb{C}$.*Object*
Definition	`=`	$\triangleq$
Judgmental equality		$=$
Propositional equality		$\equiv$

Chapter 2

Cubical Agda

2.1 Propositional equality

Judgmental equality in Agda is a feature of the type system. It is something that can be checked automatically by the type checker: In the example from the introduction $n + 0$ can be judged to be equal to n simply by expanding the definition of $+$.

On the other hand, propositional equality is something defined within the language itself. Propositional equality cannot be derived automatically. The normal definition of propositional equality is an inductive data type. Cubical Agda discards this type in favor of some new primitives.

Most of the source code related with this section is implemented in [7] it can be browsed in hyperlinked and syntax highlighted HTML online. The links can be found in the beginning of section §3.

2.1.1 The equality type

The usual notion of judgmental equality says that given a type $A : \mathcal{U}$ and two points hereof $a_0, a_1 : A$ we can form the type:

$$a_0 \equiv a_1 : \mathcal{U} \tag{2.1.1}$$

In Agda this is defined as an inductive data type with the single constructor *refl* that for any $a : A$ gives:

$$refl : a \equiv a \tag{2.1.2}$$

There also exist a related notion of *heterogeneous* equality which allows for equating points of different types. In this case given two types $A, B : \mathcal{U}$ and two points $a : A$, $b : B$ we can construct the type:

$$a \cong b : \mathcal{U} \tag{2.1.3}$$

This likewise has the single constructor *refl* that for any $a : A$ gives:

$$refl : a \cong a \tag{2.1.4}$$

In Cubical Agda these two notions are paralleled with homogeneous- and heterogeneous paths respectively.

2.1.2 The path type

Judgmental equality in Cubical Agda is encapsulated with the type:

$$Path : (P : \mathbb{I} \to \mathcal{U}) \to P\ 0 \to P\ 1 \to \mathcal{U} \tag{2.1.5}$$

The special type $\mathbb{I}$ is called the index set. The index set can be thought of simply as the interval on the real numbers from 0 to 1 (both inclusive). The family P over $\mathbb{I}$ will be referred to as the *path space* given some path $p : Path\ P\ a\ b$. By that token $P\ 0$ corresponds to the type at the left endpoint of p. Likewise $P\ 1$ is the type at the right endpoint. The type is called *Path* because the idea has roots in homotopy theory. The intuition is that *Path* describes paths in $\mathcal{U}$. I.e. paths between types. For a path p the expression $p\ i$ can be thought of as a *point* on this path. The index i describes how far along the path one has moved. An inhabitant of $Path\ P\ a_0\ a_1$ is a (dependent) function from the index set to the path space:

$$p : \prod_{i:\mathbb{I}} P\ i$$

This function must satisfy being judgmentally equal to a_0 at the left endpoint and equal to a_1 at the other end. I.e.:

$$p\ 0 = a_0$$
$$p\ 1 = a_1$$

The notion of *homogeneous equalities* is recovered when P does not depend on its argument. That is for $A : \mathcal{U}$ and $a_0, a_1 : A$ the homogenous equality between a_0 and a_1 is the type:

$$a_0 \equiv a_1 \triangleq Path\ (\lambda\ i \to A)\ a_0\ a_1$$

I will generally prefer to use the notation $a \equiv b$ when talking about non-dependent paths and use the notation $Path\ (\lambda\ i \to P\ i)\ a\ b$ when the path space is of particular interest.

With this definition we can recover reflexivity. That is, for any $A : \mathcal{U}$ and $a : A$:

$$\begin{aligned} &refl : a \equiv a \\ &refl \triangleq \lambda\ i \to a \end{aligned} \tag{2.1.6}$$

Here the path space is $P \triangleq \lambda\ i \to A$ and it satsifies $P\ i = A$ definitionally. So to inhabit it, is to give a path $\mathbb{I} \to A$ that is judgmentally a at either endpoint. This is satisfied by the constant path; i.e. the path that is constantly a at any index $i : \mathbb{I}$.

It is also surprisingly easy to show functional extensionality. Functional extensionality is the proposition that given a type $A : \mathcal{U}$, a family of types $B : A \to \mathcal{U}$ and functions $f, g : \prod_{a:A} B\ a$ gives:

$$funExt : \left(\prod_{a:A} f\ a \equiv g\ a\right) \to f \equiv g \tag{2.1.7}$$

So given $\eta : \prod_{a:A} f\ a \equiv g\ a$ we must give a path $f \equiv g$. That is a function $\mathbb{I} \to \prod_{a:A} B\ a$. So let $i : \mathbb{I}$ be given. We must now give an expression $\phi : \prod_{a:A} B\ a$ satisfying $\phi\ 0 \equiv f\ a$ and $\phi\ 1 \equiv g\ a$. This neccesitates that the expression must be a lambda abstraction, so let $a : A$ be given. We

can now apply a to η and get the path $\eta\ a : f\ a \equiv g\ a$. This exactly satisfies the conditions for ϕ. In conclusion 2.1.7 is inhabited by the term:

$$\mathit{funExt}\ \eta \triangleq \lambda\ i\ a \to \eta\ a\ i$$

With *funExt* in place we can now construct a path between *zeroLeft* and *zeroRight* – the functions defined in the introduction §1.1.1:

$$\begin{aligned} &p : \mathit{zeroLeft} \equiv \mathit{zeroRight} \\ &p \triangleq \mathit{funExt}\ \mathit{zrn} \end{aligned}$$

Here *zrn* is the proof from 1.1.1.

2.2 Homotopy levels

In ITT all equality proofs are identical (in a closed context). This means that, in some sense, any two inhabitants of $a \equiv b$ are "equally good". They do not have any interesting structure. This is referred to as Uniqueness of Identity Proofs (UIP). Unfortunately it is not possible to have a type theory with both univalence and UIP. Instead in cubical Agda we have a hierarchy of types with an increasing amount of homotopic structure. At the bottom of this hierarchy is the set of contractible types:

$$\begin{aligned} &\mathit{isContr} && : \mathcal{U} \to \mathcal{U} \\ &\mathit{isContr}\ A && \triangleq \sum_{c:A} \prod_{a:A} a \equiv c \end{aligned} \tag{2.2.1}$$

The first component of *isContr A* is called "the center of contraction". Under the propositions-as-types interpretation of type theory *isContr A* can be thought of as "the true proposition A". And indeed $\top$ is contractible:

$$(\mathit{tt}, \lambda\ x \to \mathit{refl}) : \mathit{isContr}\ \top$$

It is a theorem that if a type is contractible, then it is isomorphic to the unit-type.

The next step in the hierarchy is the set of mere propositions:

$$\begin{aligned} &\mathit{isProp} && : \mathcal{U} \to \mathcal{U} \\ &\mathit{isProp}\ A && \triangleq \prod_{a_0, a_1 : A} a_0 \equiv a_1 \end{aligned} \tag{2.2.2}$$

One can think of *isProp A* as the set of true and false propositions. And indeed both $\top$ and $\bot$ are propositions:

$$\begin{aligned} (\lambda\ \mathit{tt}, \mathit{tt} \to \mathit{refl}) &: \mathit{isProp}\ \top \\ \lambda\ \emptyset\ \emptyset &: \mathit{isProp}\ \bot \end{aligned}$$

The term $\emptyset$ is used here to denote an impossible pattern. It is a theorem that if a mere proposition A is inhabited, then so is it contractible. If it is not inhabited it is equivalent to the empty-type (or false proposition).

I will refer to a type $A : \mathcal{U}$ as a *mere proposition* if I want to stress that we have *isProp A*.

The next step in the hierarchy is the set of homotopical sets:

$$\begin{aligned} &\mathit{isSet} && : \mathcal{U} \to \mathcal{U} \\ &\mathit{isSet}\ A && \triangleq \prod_{a_0, a_1 : A} \mathit{isProp}\ (a_0 \equiv a_1) \end{aligned} \tag{2.2.3}$$

I will not give an example of a set at this point. It turns out that proving e.g. *isProp* $\mathbb{N}$ directly is not so straightforward (see [5, §3.1.4]). Hedberg's theorem states that any type with decidable equality is a set. There will be examples of sets later in this report. At this point it should be noted that the term "set" is somewhat conflated; there is the notion of sets from set theory. In Agda types are denoted `Set`. I will use it consistently to refer to a type A as a set exactly if *isSet* A is a proposition.

As the reader may have guessed the next step in the hierarchy is the type:

$$\begin{aligned} &\mathit{isGroupoid} && : \mathcal{U} \to \mathcal{U} \\ &\mathit{isGroupoid}\ A && \triangleq \prod_{a_0, a_1 : A} \mathit{isSet}\ (a_0 \equiv a_1) \end{aligned} \tag{2.2.4}$$

So it continues. In fact we can generalize this family of types by indexing them with a natural number. For historical reasons, though, the bottom of the hierarchy, the contractible types, is said to be a *-2-type*, propositions are *-1-types*, (homotopical) sets are *0-types* and so on…

Just as with paths, homotopical sets are not at the center of focus for this thesis. But I mention here some properties that will be relevant for this exposition:

Proposition: Homotopy levels are cumulative. That is, if $A : \mathcal{U}$ has homotopy level n then so does it have $n + 1$.

For any level n it is the case that to be of level n is a mere proposition.

2.3 A few lemmas

Rather than getting into the nitty-gritty details of Agda I venture to take a more "combinator-based" approach. That is I will use theorems about paths that have already been formalized. Specifically the results come from the Agda library `cubical` ([7]). I have used a handful of results from this library as well as contributed a few lemmas myself[1].

These theorems are all purely related to homotopy type theory and as such not specific to the formalization of Category Theory. I will present a few of these theorems here as they will be used throughout chapter 3. They should also give the reader some intuition about the path type.

2.3.1 Path induction

The induction principle for paths intuitively gives us a way to reason about a type family indexed by a path by only considering if said path is *refl* (the *base case*). For *based path induction*, that equality is *based* at some element $a : A$.

[1] The module `Cat.Prelude` lists the upstream dependencies. My contribution to `cubical` can as well be found in the git logs which are available at `https://github.com/Saizan/cubical-demo` .

Let a type $A : \mathcal{U}$ and an element of the type $a : A$ be given. a is said to be the base of the induction. Given a family of types:

$$D : \prod_{b:A} \prod_{p:a\equiv b} \mathcal{U}$$

and an inhabitant of D at *refl*:

$$d : D\ a\ refl$$

We have the function:

$$pathJ\ D\ d : \prod_{b:A} \prod_{p:a\equiv b} D\ b\ p \tag{2.3.1}$$

A simple application of *pathJ* is for proving that *sym* is an involution. Namely for any set $A : \mathcal{U}$, points $a, b : A$ and a path between them $p : a \equiv b$:

$$sym\ (sym\ p) \equiv p \tag{2.3.2}$$

The proof will be by induction on p and will be based at a. That is D will be the family:

$$D : \prod_{b':A} \prod_{p:a\equiv b'} \mathcal{U}$$
$$D\ b'\ p' \triangleq sym\ (sym\ p') \equiv p'$$

The base case will then be:

$$d : sym\ (sym\ refl) \equiv refl$$
$$d \triangleq refl$$

The reason *refl* proves this is that $sym\ refl = refl$ holds definitionally. In summary 2.3.2 is inhabited by the term:

$$pathJ\ D\ d\ b\ p : sym\ (sym\ p) \equiv p$$

Another application of *pathJ* is for proving associativity of *trans*. That is, given a type $A : \mathcal{U}$, elements of A, $a, b, c, d : A$ and paths between them $p : a \equiv b$, $q : b \equiv c$ and $r : c \equiv d$ we have the following:

$$trans\ p\ (trans\ q\ r) \equiv trans\ (trans\ p\ q)\ r \tag{2.3.3}$$

In this case the induction will be based at c (the left-endpoint of r) and over the family:

$$T : \prod_{d':A} \prod_{r':c\equiv d'} \mathcal{U}$$
$$T\ d'\ r' \triangleq trans\ p\ (trans\ q\ r') \equiv trans\ (trans\ p\ q)\ r'$$

The base case is proven with t which is defined as:

$$\begin{aligned} trans\ p\ (trans\ q\ refl) &\equiv trans\ p\ q \\ &\equiv trans\ (trans\ p\ q)\ refl \end{aligned}$$

Here we have used the proposition $trans\ p\ refl \equiv p$ without proof. In conclusion 2.3.3 is inhabited by the term:

$$pathJ\ T\ t\ d\ r$$

We shall see another application of path induction in 3.3.5.

2.3.2 Paths over propositions

Another very useful combinator is *lemPropF*: Given a type $A : \mathcal{U}$ and a type family on A; $D : A \to \mathcal{U}$. Let $propD : \prod_{x:A} isProp\ (D\ x)$ be the proof that D is a mere proposition for all elements of A. Furthermore say we have a path between some two elements in A; $p : a_0 \equiv a_1$ then we can built a heterogeneous path between any two elements of $d_0 : D\ a_0$ and $d_1 : D\ a_1$.

$$lemPropF\ propD\ p : Path\ (\lambda\ i \to D\ (p\ i))\ d_0\ d_1$$

Note that d_0 and d_1, though points of the same family, have different types. This is quite a mouthful, so let me try to show how this is a very general and useful result.

Often when proving equalities between elements of some dependent types *lemPropF* can be used to boil this complexity down to showing that the dependent parts of the type are mere propositions. For instance say we have a type:

$$T \triangleq \sum_{a:A} D\ a$$

for some proposition $D : A \to \mathcal{U}$. That is we have $propD : \prod_{a:A} isProp\ (D\ a)$. If we want to prove $t_0 \equiv t_1$ for two elements $t_0, t_1 : T$ then this will be a pair of paths:

$$\begin{aligned} p :& fst\ t_0 \equiv fst\ t_1 \\ & Path\ (\lambda\ i \to D\ (p\ i))\ (snd\ t_0)\ (snd\ t_1) \end{aligned}$$

Here *lemPropF* directly allow us to prove the latter of these given that we have already provided p.

$$lemPropF\ propD\ p : Path\ (\lambda\ i \to D\ (p\ i))\ (snd\ t_0)\ (snd\ t_1)$$

2.3.3 Functions over propositions

$\prod$-types preserve propositionality when the co-domain is always a proposition.

$$propPi : \left(\prod_{a:A} isProp\ (P\ a)\right) \to isProp\ \left(\prod_{a:A} P\ a\right)$$

2.3.4 Pairs over propositions

$\sum$-types preserve propositionality whenever its first component is a proposition and its second component is a proposition for all points of the left type.

$$propSig : isProp\ A \to \left(\prod_{a:A} isProp\ (P\ a)\right) \to isProp\ \left(\sum_{a:A} P\ a\right)$$

Chapter 3

Category Theory

The source code for this formalization, including this report, is available as open source software at:

https://github.com/fredefox/cat/

All modules imported for this formalization can be browsed at this link[1]:

http://web.student.chalmers.se/~hanghj/cat/doc/html/

The concepts formalized in this development are:

Name	Module
Equivalences	Cat.Equivalence
Categories	Cat.Category
Functors	Cat.Category.Functor
Products	Cat.Category.Product
Exponentials	Cat.Category.Exponential
Cartesian closed categories	Cat.Category.CartesianClosed
Natural transformations	Cat.Category.NaturalTransformation
Yoneda embedding	Cat.Category.Yoneda
Monads	Cat.Category.Monad
Kleisli Monads	Cat.Category.Monad.Kleisli
Monoidal Monads	Cat.Category.Monad.Monoidal
Voevodsky's construction	Cat.Category.Monad.Voevodsky
Opposite category	Cat.Categories.Opposite
Category of sets	Cat.Categories.Sets
Span category	Cat.Categories.Span

[1] In case the linked sources are unavailable the html documentation can be generated by navigating to the root directory of the project and executing `make html`.

Furthermore the following items have been partly formalized:

Name	Module
Category of categories	`Cat.Categories.Cat`
Category of relations	`Cat.Categories.Rel`
Category of functors	`Cat.Categories.Fun`
Free category	`Cat.Categories.Free`
Monoids	`Cat.Category.Monoid`

I also provide a various results about these. E.g. I have shown that the category of sets has products. In the following I aim to demonstrate some of the techniques employed in this formalization. In the interest of brevity I will not detail all the things I have formalized. Instead I have selected parts of this formalization that highlight some interesting proof techniques relevant to doing proofs in Cubical Agda. This chapter will focus on the definition of *categories*, *equivalences*, the *opposite category*, the *category of sets*, *products*, the *span category* and the two formulations of *monads*.

One technique employed throughout this formalization is the idea of distinguishing types with more or less homotopical structure. To do this I have followed the following design-principle: I have split concepts up into things that represent *data* and *laws* about this data. The idea is that we can provide a proof that the laws are mere propositions. As an example a category is defined to have two members: *raw* which is a collection of the data and *isCategory* which asserts some laws about that data.

This allows me to reason about things in a more "standard mathematical way", where one can reason about two categories by simply focusing on the data. This is achieved by creating a function embodying the equality principle for a given type.

For the rest of this chapter I will present some of these results. For didactic reasons no source-code has been included in this chapter. To see the formal definitions the reader is referred to the implementation which is linked in the tables above.

3.1 Categories

The data for a category consist of a type for the sort of objects, a type for the sort of arrows, an identity arrow and a composition operation for arrows. Another record encapsulates some laws about this data: associativity of composition, identity law for the identity morphism. These are standard constituents of a category and can be found in typical mathematical expositions on the topic. We shall impose one further requirement on what it means to be a category, namely that the type of arrows form a set.

Categories whose type of arrows form a set are called *1-categories*. It is possible to relax this requirement. This would lead to the notion of higher categories ([5, p. 307]). However this thesis will restrict itself to 1-categories. Generalizing this work to higher categories would be a very natural extension of this work.

Raw categories satisfying all of the above requirements are called a *pre-categories*. As a further requirement to be a proper category we require the arrows to be univalent. Before we can define this, I must introduce two additional definitions: If we let p be a witness to the identity law,

which formally is:

$$IsIdentity \triangleq \prod_{A,B:Object} \prod_{f:Arrow\ A\ B} (id \lll f \equiv f) \times (f \lll id \equiv f) \tag{3.1.1}$$

Then we can construct the identity isomorphism $idIso : (identity, identity, p) : A \cong A$ for any object A. Here $\cong$ denotes isomorphism on objects (whereas $\cong$ denotes isomorphism on types). This will be elaborated further on in sections §3.2 and §3.3. Moreover due to substitution for paths we can construct an isomorphism from *any* path:

$$idToIso : A \equiv B \to A \cong B \tag{3.1.2}$$

The univalence criterion for categories states that *idToIso* must be an equivalence. The requirement is similar to univalence for types, but where isomorphism on objects play the role of equivalence on types. Formally:

$$isEquiv\ (A \equiv B)\ (A \cong B)\ idToIso \tag{3.1.3}$$

Note that 3.1.3 is *not* the same as:

$$(A \equiv B) \simeq (A \cong B) \tag{3.1.4}$$

However the two are logically equivalent: one can construct the latter from the former simply by "forgetting" that *idToIso* plays the role of the equivalence. The other direction is more involved and will be discussed in section §3.3.

In summary the definition of a category is the following collection of data:

$$Object : \mathcal{U} \tag{3.1.5}$$

$$Arrow : Object \to Object \to \mathcal{U} \tag{3.1.6}$$

$$identity : Arrow\ A\ A \tag{3.1.7}$$

$$\lll : Arrow\ B\ C \to Arrow\ A\ B \to Arrow\ A\ C \tag{3.1.8}$$

and laws:

$$h \lll (g \lll f) \equiv (h \lll g) \lll f \tag{3.1.9}$$

$$(identity \lll f \equiv f) \times (f \lll identity \equiv f) \tag{3.1.10}$$

$$isSet\ (Arrow\ A\ B) \tag{3.1.11}$$

$$isEquiv\ (A \equiv B)\ (A \cong B)\ idToIso \tag{3.1.3}$$

The function $\lll$ denotes arrow composition (right-to-left) and reverse function composition (left-to-right, diagrammatic order) is denoted $\ggg$. The objects (A, B and C) and arrows (f, g, h) are implicitly universally quantified.

With all this in place it is now possible to prove that all the laws are indeed mere propositions. Most of the proofs simply use the fact that the type of arrows are sets. This is because most of the laws are a collection of equations between arrows in the category. And since such a proof does not have any content, exactly because the type of arrows form a set, two witnesses must be the same. All the proofs are really quite mechanical. Let us have a look at one of them: Proving that 3.1.1 is a mere proposition:

$$isProp\ IsIdentity \tag{3.1.12}$$

There are multiple ways to do this. Perhaps one of the more intuitive proofs is by way of the 'combinators' *propPi* and *propSig* presented in sections §2.3.3 and §2.3.4:

$$propPi : \left(\prod_{a:A} isProp\ (P\ a)\right) \to isProp\ \left(\prod_{a:A} P\ a\right)$$

$$propSig : isProp\ A \to \left(\prod_{a:A} isProp\ (P\ a)\right) \to isProp\ \left(\sum_{a:A} P\ a\right)$$

The proof goes like this: We 'eliminate' the 3 function abstractions by applying *propPi* three times. So our proof obligation becomes:

$$isProp\ ((id \circ f \equiv f) \times (f \circ id \equiv f))$$

We then eliminate the (non-dependent) sigma-type by applying *propSig* giving us the two obligations $isProp\ (id \circ f \equiv f)$ and $isProp\ (f \circ id \equiv f)$ which follows from the type of arrows being a set.

This example illustrates nicely how we can use these combinators to reason about 'canonical' types like $\sum$ and $\prod$. Similar combinators can be defined at the other homotopic levels. These combinators are however not applicable in situations where we want to reason about other types e.g. types we have defined ourselves. For instance, after we have proven that all the projections of pre-categories are propositions, we would like to bundle this up to show that the type of pre-categories is also a proposition. Formally:

$$isProp\ IsPreCategory \tag{3.1.13}$$

Where the definition of *IsPreCategory* is the triple:

$$\begin{aligned} isAssociative &: IsAssociative \\ isIdentity &: IsIdentity \\ arrowsAreSets &: ArrowsAreSets \end{aligned}$$

Each corresponding to the first three laws for categories. Note that since *IsPreCategory* is not formulated with a chain of sigma-types we will not have any combinators available to help us here. Instead the path type must be used directly.

The type 3.1.13 is judgmentally the same as:

$$\prod_{a,b:IsPreCategory} a \equiv b$$

To prove the proposition let $a, b : IsPreCategory$ be given. To prove the equality $a \equiv b$ is to give a continuous path from the index-type into the path space; i.e. a function $\mathbb{I} \to IsPreCategory$. This path must satisfy being being judgmentally the same as a at the left endpoint and b at the right endpoint. We know we can form a continuous path between all projections of a and b. This follows from the type of all the projections being mere propositions. For instance, the path between $a.isIdentity$ and $b.isIdentity$ is simply formed by:

$$propIsIdentity\ a.isIdentity\ b.isIdentity : a.isIdentity \equiv b.isIdentity$$

To give the continuous function $\mathbb{I} \to IsPreCategory$, which is our goal, we introduce $i : \mathbb{I}$ and proceed by constructing an element of *IsPreCategory* by using the fact that all the projections are propositions to generate paths between all projections. Once we have such a path e.g. $p : a.isIdentity \equiv b.isIdentity$ we can eliminate it with i and thus obtain $p\ i : (p\ i).isIdentity$. This element satisfies exactly that it corresponds to the corresponding projections at either endpoint. Thus the element we construct at i becomes the triple:

$$\begin{array}{llll} propIsAssociative & a.isAssociative & b.isAssociative & i \\ propIsIdentity & a.isIdentity & b.isIdentity & i \\ propArrowsAreSets & a.arrowsAreSets & b.arrowsAreSets & i \end{array} \tag{3.1.14}$$

I have found this to be a general pattern when proving things in homotopy type theory, namely that you have to wrap and unwrap equalities at different levels. It is worth noting that proving this theorem with the regular inductive equality type would already not be possible since we

at least need functional extensionality (the projections are all $\prod$-types). Assuming we had functional extensionality available to us as an axiom, we would use functional extensionality to retrieve the equalities in a and b; pattern match on them to see that they are both *refl* and then close the proof with *refl*. Of course this theorem is not so interesting in the setting of ITT since we know a priori that equality proofs are unique.

The situation is a bit more complicated when we have a dependent type. For instance when we want to show that *IsCategory* is a mere proposition. The type *IsCategory* is a record with two fields: a witness of being a pre-category and the univalence condition. Recall that the univalence condition is indexed by the identity-proof. To follow the same recipe as above, let $a, b : \mathit{IsCategory}$ be given, to show them equal, we now need to give two paths. One homogeneous:

$$p : a.\mathit{isPreCategory} \equiv b.\mathit{isPreCategory}$$

and one heterogeneous:

$$\mathit{Path}\ (\lambda\ i \to (p\ i).\mathit{Univalent})\ a.\mathit{isPreCategory}\ b.\mathit{isPreCategory}$$

This path depends on the choice of p. The first of these we can provide since, as we have shown, *IsPreCategory* is a proposition. However, even though *Univalent* is also a proposition, we cannot use this directly to show the latter. This is because *isProp* talks about non-dependent paths. So we need to 'promote' the result that univalence is a proposition to a heterogeneous path. To this end we can use *lemPropF*, which was introduced in §2.3.2.

Looking at the definition of *lemPropF* we have that $A = \mathit{IsIdentity}\ \mathit{identity}$ and $D = \mathit{Univalent}$ to give the path at hand. We have shown that being a category is a proposition, a result that holds for any choice of identity proof so it will also hold for the witness obtained at an arbitrary point along p. Finally we must provide a proof that the identity proofs at a and b are indeed the same, this we can extract from p by applying congruence of paths:

$$\mathit{cong}\ \mathit{isIdentity}\ p$$

In summary the heterogeneous path is inhabited by:

$$\mathit{lemPropF}\ \mathit{propUnivalent}\ (\mathit{cong}\ p.\mathit{isIdentity})$$

This finishes the proof that being-a-category is a mere proposition (3.1.13).

When we have a proper category, we can make precise the notion of "identifying isomorphic types". That is, we can construct the function:

$$\mathit{isoToId} : (A \cong B) \to (A \equiv B)$$

A perhaps somewhat surprising application of this is that we can show that terminal objects are propositional:

$$\mathit{isProp}\ \mathit{Terminal} \tag{3.1.15}$$

It follows from the usual observation that any two terminal objects are isomorphic - and since categories are univalent, so are they equal. The proof is omitted here, but the curious reader can check the implementation for the details. It is in the module:

`Cat.Category` ↗

3.2 Equivalences

The usual notion of a function $f : A \to B$ having an inverses is:

$$\sum_{g:B\to A} (f \circ g \equiv identity) \times (g \circ f \equiv identity) \tag{3.2.1}$$

This is defined in [5, p. 129] where it is referred to as the a "quasi-inverse". We shall refer to the type 3.2.1 as *Isomorphism* f. This also gives rise to the following type:

$$A \cong B \triangleq \sum_{f:A\to B} Isomorphism\ f \tag{3.2.2}$$

At the same place [5] gives an "interface" for what the judgment $isEquiv : (A \to B) \to \mathcal{U}$ must provide:

$$fromIso : Isomorphism\ f \to isEquiv\ f \tag{3.2.3}$$
$$toIso : isEquiv\ f \to Isomorphism\ f \tag{3.2.4}$$
$$isEquiv\ f \tag{3.2.5}$$

The maps *fromIso* and *toIso* naturally extend to these maps:

$$fromIsomorphism : A \cong B \to A \simeq B \tag{3.2.6}$$
$$toIsomorphism : A \simeq B \to A \cong B \tag{3.2.7}$$

Having this interface gives us both a way to think rather abstractly about how to work with equivalences and a way to use ad hoc definitions of equivalences. The specific instantiation of *isEquiv* as defined in [6] is:

$$isEquiv\ f \triangleq \prod_{b:B} isContr\ (fiber\ f\ b)$$

where

$$fiber\ f\ b \triangleq \sum_{a:A} (b \equiv f\ a)$$

I give its definition here mainly for completeness, because as I stated we can move away from this specific instantiation and think about it more abstractly once we have shown that this definition actually works as an equivalence.

The implementation of *fromIso* can be found in [6] where it is known as *gradLemma*. The implementation of *fromIso* as well as the proof that this equivalence is a proposition (3.2.5) can be found in my implementation.

We say that two types $A\ B : \mathcal{U}$ are equivalent exactly if there exists an equivalence between them:

$$A \simeq B \triangleq \sum_{f:A\to B} isEquiv\ f \tag{3.2.8}$$

Note that the term equivalence here is overloaded referring both to the map $f : A \to B$ and the type $A \simeq B$. The notion of an isomorphism is similarly conflated as isomorphism can refer to the type $A \cong B$ as well as the the map $A \to B$ that witness this. I will use these conflated terms when it is clear from the context what is being referred to.

Both $\cong$ and $\simeq$ form equivalence relations (no pun intended).

3.3 Univalence

As noted in the introduction the univalence for types $A\ B : \mathcal{U}$ states that:

$$Univalence \triangleq (A \equiv B) \simeq (A \simeq B)$$

As mentioned the univalence criterion for some category $\mathbb{C}$ says that for all *objects* $A\ B$ we must have:

$$isEquiv\ (A \equiv B)\ (A \cong B)\ idToIso$$

This is logically equivalent to

$$(A \equiv B) \simeq (A \cong B)$$

Given that we saw in the previous section that we can construct an equivalence from an isomorphism it suffices to demonstrate:

$$(A \equiv B) \cong (A \cong B)$$

That is, we must demonstrate that there is an isomorphism (on types) between equalities and isomorphisms (on arrows). It is worthwhile to dwell on this for a few seconds. This type looks very similar to univalence for types and is therefore perhaps a bit more intuitive to grasp the implications of. Of course univalence for types (which is a theorem – i.e. provably holds) does not imply univalence of all pre-category since morphisms in a category are not regular functions, instead they can be thought of as a generalization hereof. The univalence criterion therefore is simply a way of restricting arrows to behave like regular functions with respects to paths.

I will now mention a few helpful theorems that follow from univalence that will become useful later.

Obviously univalence gives us an isomorphism between $A \equiv B$ and $A \cong B$. I will name these for convenience:

$$idToIso : A \equiv B \to A \cong B$$

$$isoToId : A \cong B \to A \equiv B$$

The next few theorems are variations on theorem 9.1.9 from [5]. Let an isomorphism $A \cong B$ in some category $\mathbb{C}$ be given. Name the isomorphism $\iota : A \to B$ and its inverse $\iota^{-1} : B \to A$. Since $\mathbb{C}$ is a category (and therefore univalent) the isomorphism induces a path

$$p \triangleq idToIso\ (\iota, \iota^{-1}, ...) : A \equiv B$$

From this equality we can get two further paths:

$$p_{dom} : Arrow\ A\ X \equiv Arrow\ B\ X$$
$$p_{cod} : Arrow\ X\ A \equiv Arrow\ X\ B$$

We then have the following two theorems:

$$coeDom : \prod_{f : A \to X} coe\ p_{dom}\ f \equiv f \lll \iota^{-1} \tag{3.3.1}$$

$$coeCod : \prod_{f : A \to X} coe\ p_{cod}\ f \equiv \iota \lll f \tag{3.3.2}$$

I will give the proof of the first theorem here, the second one is analogous.

$$\begin{aligned} coe\ p_{dom}\ f &\equiv f \lll (idToIso\ p)_2 && \text{lemma} \\ &\equiv f \lll \iota^{-1} && idToIso \text{ and } isoToId \text{ are inverses} \end{aligned}$$

In the second step we use the fact that p is constructed from the isomorphism ι. The subscript in term $(idToIso\ p)_2$ is intended to denote the inverse map $B \to A$ from the isomorphism $idToIso\ p : A \cong B$. The helper-lemma is similar to what we are trying to prove but talks about paths rather than isomorphisms:

$$\prod_{f:Arrow\ A\ B} \prod_{p:A \equiv B} coe\ p_{dom}\ f \equiv f \lll (idToIso\ p)_2 \tag{3.3.3}$$

Again p_{dom} denotes the path $Arrow\ A\ X \equiv Arrow\ B\ X$ induced by p. To prove this statement let f and p be given then we invoke based path induction. The induction will be based at $A : Object$. Let $\widetilde{B} : Object$ and $\tilde{p} : A \equiv \widetilde{B}$ be given. The family that we perform induction over will be:

$$D\ \widetilde{B}\ \tilde{p} \triangleq coe\ \tilde{p}_{dom}\ f \equiv f \lll (idToIso\ \tilde{p})_2 \tag{3.3.4}$$

The base-case therefore becomes $d : coe\ refl_{dom}\ f \equiv f \lll (idToIso\ refl)_2$ and is inhabited by:

$$\begin{aligned}
coe\ refl_{dom}\ f &\equiv f && refl \text{ is a neutral element for } coe \\
&\equiv f \lll identity \\
&\equiv f \lll subst\ refl\ identity && refl \text{ is a neutral element for } subst \\
&= f \lll (idToIso\ refl)_2 && \text{By definition of } idToIso
\end{aligned}$$

To close the based-path-induction we must supply the value "at the other end". In this case this is simply $B : Object$ and $p : A \equiv B$ which we have. In summary the proof of 3.3.3 is the term:

$$pathJ\ D\ d\ B\ p \tag{3.3.5}$$

This finishes the proof of 3.3.3 and thus 3.3.1.

3.4 Categories

3.4.1 Opposite category

The first category I will present is a pure construction on categories. Given some category we can construct its dual, called the opposite category. Starting with a simple example allows us to focus on how we work with equivalences and univalence rather than being distracted by some intricate structure of the category.

Let $\mathbb{C}$ be some category, we then define the opposite category $\mathbb{C}^{Op}$. It has the same objects, but the type of arrows are flipped, that is to say an arrow from A to B in the opposite category corresponds to an arrow from B to A in the underlying category. The identity arrow is the same as the one in the underlying category (they have the same type). Function composition will be reverse function composition from the underlying category.

I will refer to things in terms of the underlying category unless they have a line above them. E.g. $idToIso$ is a function in the underlying category and the corresponding thing is denoted $\overline{idToIso}$ in the opposite category.

Showing that this forms a pre-category is rather straightforward.

$$h \ggg (g \ggg f) \equiv h \ggg g \ggg f$$

Since $\ggg$ is reverse function composition this is just the symmetric version of associativity.

$$identity \ggg f \equiv f \times f \ggg identity \equiv f$$

This is just the swapped version of identity.

Finally, that the arrows form sets just follows by flipping the order of the arguments. Or in other words: since $Arrow\ A\ B$ is a set for all $A\ B : Object$ then so is $Arrow\ B\ A$.

Now, to show that this category is univalent is not as straightforward. Luckily section §3.2 gave us some tools to work with equivalences. We saw that we can prove this category univalent by giving an inverse to $\overline{idToIso} : (A \equiv B) \to (A \widetilde{\cong} B)$. From the original category we have that $idToIso : (A \equiv B) \to (A \cong B)$ is an isomorphism. Let us denote its inverse with $isoToId : (A \cong B) \to (A \equiv B)$. If we squint we can see what we need is a way to go between $\overline{\cong}$ and $\cong$.

An inhabitant of $A \cong B$ is simply an arrow $f : Arrow\ A\ B$ and its inverse $g : Arrow\ B\ A$ (and a witness to them being inverses). In the opposite category g will play the role of the isomorphism and f will be the inverse. Similarly we can go in the opposite direction. I name these maps $shuffle : (A \cong B) \to (A \widetilde{\cong} B)$ and $shuffle^{-1} : (A \widetilde{\cong} B) \to (A \cong B)$ respectively.

As the inverse of $\overline{idToIso}$ I will pick $\overline{isoToId} \triangleq isoToId \circ shuffle$. The proof that they are inverses goes as follows:

$$\begin{aligned}
\overline{isoToId} \circ \overline{idToIso} &= isoToId \circ shuffle \circ \overline{idToIso} \\
&\equiv isoToId \circ shuffle \circ shuffle^{-1} \circ idToIso && \text{lemma} \\
&\equiv isoToId \circ idToIso && shuffle \text{ is an isomorphism} \\
&\equiv identity && isoToId \text{ is an isomorphism}
\end{aligned}$$

The other direction is analogous.

The lemma used in step 2 of this proof states that $\overline{idToIso} \equiv shuffle^{-1} \circ idToIso$. This is a rather straightforward proof since being-an-inverse-of is a proposition, so it suffices to show that their first components are equal but this holds judgmentally.

This concludes the proof that the opposite category is in fact a category. Now, to prove that opposite-of is an involution we must show:

$$\prod_{\mathbb{C} : Category} \left(\mathbb{C}^{Op}\right)^{Op} \equiv \mathbb{C}$$

The laws in $\left(\mathbb{C}^{Op}\right)^{Op}$ get quite involved. Luckily since being-a-category is a mere proposition, we need not concern ourselves with this bit when proving the above. We can use the equality principle for categories that let us prove an equality just by giving an equality on the data-part. So, given a category $\mathbb{C}$ all we must provide is the following proof:

$$raw\ \left(\mathbb{C}^{Op}\right)^{Op} \equiv raw\ \mathbb{C}$$

And these are judgmentally the same. I remind the reader that the left-hand side is constructed by flipping the arrows, which judgmentally is an involution.

3.4.2 Category of sets

The category of sets has as objects, not types, but only those types that are homotopic sets. This is encapsulated in Agda with the following type:

$$Set \triangleq \sum_{A:\mathcal{U}} isSet\ A$$

The more straightforward notion of a category where the objects are types is not a valid (1-)category. This stems from the fact that types in cubical Agda can have higher homotopic structure.

Univalence does not follow immediately from univalence for types:

$$(A \equiv B) \simeq (A \simeq B)$$

because here $A, B : \mathcal{U}$, whereas the objects in this category have the type *Set* so we cannot form the type $hA \simeq hB$ for objects $hA\ hB : Set$. Instead I show that this category satisfies:

$$(hA \equiv hB) \simeq (hA \cong hB)$$

Which, as we saw in section §3.3, is sufficient to show that the category is univalent. The way that I have shown this is with a three-step process. For objects $(A, s_A), (B, s_B) : Set$ I show the following chain of equivalences:

$$\begin{aligned} ((A, s_A) \equiv (B, s_B)) &\simeq (A \equiv B) && \text{3.4.1} \\ &\simeq (A \simeq B) && \text{Univalence} \\ &\simeq ((A, s_A) \cong (B, s_B)) && \text{3.4.2 and 3.4.3} \end{aligned}$$

Since $\simeq$ is an equivalence relation we can chain these equivalences together. Step one will be proven with the lemma:

$$\left(\prod_{a:A} isProp\ (P\ a)\right) \to \prod_{x\ y:\sum_{a:A} P\ a} (x \equiv y) \simeq (fst\ x \equiv fst\ y) \tag{3.4.1}$$

The lemma states that for pairs whose second component are mere propositions equality is equivalent to equality of the first components. In this case the type-family P is *isSet* which itself is a proposition for any type $A : \mathcal{U}$. Step two is univalence for types. Step three will be proven with the following lemma:

$$\prod_{a:A} (P\ a \simeq Q\ a) \to \sum_{a:A} P\ a \simeq \sum_{a:A} Q\ a \tag{3.4.2}$$

which says that if two type-families are equivalent at all points, then pairs with identical first components and these families as second components will also be equivalent. For our purposes $P \triangleq isEquiv\ A\ B$ and $Q \triangleq Isomorphism$. We must finally prove:

$$\prod_{f:A\to B} (isEquiv\ A\ B\ f \simeq Isomorphism\ f) \tag{3.4.3}$$

Lets us first prove 3.4.1: Let $propP : \prod_{a:A} isProp\ (P\ a)$ and $x\ y : \sum_{a:A} P\ a$ be given. Because of *fromIsomorphism* it suffices to give an isomorphism between $x \equiv y$ and $fst\ x \equiv fst\ y$:

$$\begin{aligned} f &\triangleq cong\ fst && : x \equiv y && \to fst\ x \equiv fst\ y \\ g &\triangleq lemSig\ propP\ x\ y && : fst\ x \equiv fst\ y && \to x \equiv y \end{aligned}$$

Here *lemSig* is a lemma that says that if the second component of a pair is a proposition it suffices to give a path between its first components to construct an equality of the two pairs:

$$lemSig : \left(\prod_{x:A} isProp\ (B\ x)\right) \to \prod_{u\ v:\sum_{a:A} B\ a} (fst\ u \equiv fst\ v) \to u \equiv v$$

The proof that these are indeed inverses of one another has been omitted. The details can be found in the module:

`Cat.Categories.Sets`

Now to prove 3.4.2: Let $e : \prod_{a:A}(P\ a \simeq Q\ a)$ be given. To prove the equivalence it suffices to give an isomorphism between $\sum_{a:A} P\ a$ and $\sum_{a:A} Q\ a$, but since they have identical first components it suffices to give an isomorphism between $P\ a$ and $Q\ a$ for all $a : A$. This is exactly what we can get from the equivalence e.

Lastly we prove 3.4.3. Let $f : A \to B$ be given. For the maps we choose:

$$toIso : isEquiv\ f \to Isomorphism\ f$$
$$fromIso : Isomorphism\ f \to isEquiv\ f$$

As mentioned in section §3.2. These maps are not in general inverses of each other. Instead, we will use the fact that A and B are sets. The first thing we must prove is:

$$fromIso \circ toIso \equiv identity_{isEquiv\ f}$$

For this we can use the fact that being-an-equivalence is a mere proposition. For the other direction:

$$toIso \circ fromIso \equiv identity_{Isomorphism\ f}$$

We will show that *Isomorphism f* is also a mere proposition. To this end, let $X, Y : Isomorphism\ f$ be given. Name the maps $x, y : B \to A$ respectively. Now, the proof that X and Y are the same is a pair of paths: $p : x \equiv y$ and $Path\ (\lambda\ i \to AreInverses\ f\ (p\ i))\ \mathcal{X}\ \mathcal{Y}$ where $\mathcal{X}$ and $\mathcal{Y}$ denotes the witnesses that x (respectively y) is an inverse to f. The path p is inhabited by:

$$\begin{aligned}
x &= x \circ identity \\
&\equiv x \circ (f \circ y) && y \text{ is an inverse to } f \\
&\equiv (x \circ f) \circ y \\
&\equiv identity \circ y && x \text{ is an inverse to } f \\
&= y
\end{aligned}$$

For the other (dependent) path we can prove that being-an-inverse-of is a proposition and then use *lemPropF*. To this end we prove the generalization:

$$\prod_{g:B\to A} isProp\ (AreInverses\ f\ g) \tag{3.4.4}$$

but $AreInverses\ f\ g$ is a pair of equations on arrows, so we use *propSig* and the fact that both A and B are sets to close this proof.

3.5 Products

In the following a technique for using categories to prove properties will be demonstrated. The goal in this section is to show that products are propositions:

$$\prod_{\mathbb{C}:Category} \prod_{A\ B:Object} isProp\ (Product\ \mathbb{C}\ A\ B)$$

where $Product\ \mathbb{C}\ A\ B$ denotes the type of products of objects A and B in the category $\mathbb{C}$. I do this by constructing a category whose terminal objects are equivalent to products in $\mathbb{C}$. Since terminal objects are propositional in a proper category and equivalences preserve homotopy level, then we know that products are also propositions. But before we get to that, we recall the definition of products.

3.5.1 Definition of products

Given a category $\mathbb{C}$ and two objects A and B in $\mathbb{C}$ we say that a product is triple consisting of an object and a pair of arrows. The object is denoted with $A \times B$ in $\mathbb{C}$ and is also referred to as the product (object) of A and B. The arrows will be named $\pi_1 : A \times B \to A$ and $\pi_2 : A \times B \to B$ also called the projections of the product. The projections must satisfy the following property:

For all $X : Object$, $f : Arrow\ X\ A$ and $g : Arrow\ X\ B$ we have that there exists a unique arrow $\pi : Arrow\ X\ (A \times B)$ satisfying

$$\pi_1 \lll \pi \equiv f \times \pi_2 \lll \pi \equiv g \tag{3.5.1}$$

The arrow π is called the product (arrow) of f and g.

3.5.2 Span category

Given a base category $\mathbb{C}$ and two objects in this category $\mathcal{A}$ and $\mathcal{B}$ we construct the *span category*. The type of objects in this category shall be an object in the underlying category, X and two arrows (also from the underlying category) $Arrow\ X\ \mathcal{A}$ and $Arrow\ X\ \mathcal{B}$.

An arrow between objects $A,\ a_{\mathcal{A}},\ a_{\mathcal{B}}$ and $B,\ b_{\mathcal{A}},\ b_{\mathcal{B}}$ in this category will consist of an arrow from the underlying category $f : Arrow\ A\ B$ satisfying:

$$b_{\mathcal{A}} \lll f \equiv a_{\mathcal{A}} \times b_{\mathcal{B}} \lll f \equiv a_{\mathcal{B}} \tag{3.5.2}$$

The identity morphism is the identity morphism from the underlying category. This choice satisfies 3.5.2 because of the right-identity law from the underlying category.

For composition of arrows $f : Arrow\ A\ B$ and $g : Arrow\ B\ C$ we choose $g \lll f$ and we must now verify that it satisfies 3.5.2:

$$\begin{aligned} c_{\mathcal{A}} \lll (f \lll g) &\equiv (c_{\mathcal{A}} \lll f) \lll g && \text{Associativity} \\ &\equiv b_{\mathcal{A}} \lll g && f \text{ satisfies 3.5.2} \\ &\equiv a_{\mathcal{A}} && g \text{ satisfies 3.5.2} \end{aligned}$$

Now we must verify the category-laws. For all the laws we will follow the pattern of using the law from the underlying category, and that the type of arrows form a set. For instance, to prove associativity we must prove that

$$\overline{h} \lll (\overline{g} \lll \overline{f}) \equiv (\overline{h} \lll \overline{g}) \lll \overline{f} \tag{3.5.3}$$

Here $\lll$ refers to the 'embellished' composition and $\overline{f}$, $\overline{g}$ and $\overline{h}$ are triples consisting of arrows from the underlying category (f, g and h) and a pair of witnesses to 3.5.2. The proof obligations consists of two things. The first one is:

$$h \lll (g \lll f) \equiv (h \lll g) \lll f \tag{3.5.4}$$

The other proof obligation is that the witness to 3.5.2 for the left-hand-side and the right-hand-side are the same.

The proof of the first goal comes directly from the underlying category. The type of the second goal is very complicated. I will not write it out in full here, but for the purpose of this exposition it will suffice to show the type of the path space. Note that the arrows in 3.5.3 are arrows between objects on the form $\overline{A} = (A, a_{\mathcal{A}}, a_{\mathcal{B}})$ to $\overline{D} = (D, d_{\mathcal{A}}, d_{\mathcal{B}})$ where $a_{\mathcal{A}}$, $a_{\mathcal{B}}$, $d_{\mathcal{A}}$ and $d_{\mathcal{B}}$ are arrows in the underlying category. Given that p is the chosen proof of 3.5.4 we then have that the witness to 3.5.2 vary over the type:

$$\lambda\ i \to d_{\mathcal{A}} \lll p\ i \equiv a_{\mathcal{A}} \times d_{\mathcal{B}} \lll p\ i \equiv a_{\mathcal{B}} \tag{3.5.5}$$

These paths are in the type of the hom-set of the underlying category, so they are mere propositions. We cannot apply the fact that arrows in $\mathbb{C}$ are sets directly, however, since *isSet* only talks about non-dependent paths, instead we generalize 3.5.5 to:

$$\prod_{f:Arrow\ X\ Y} isProp\ (y_{\mathcal{A}} \lll f \equiv x_{\mathcal{A}} \times y_{\mathcal{B}} \lll f \equiv x_{\mathcal{B}}) \tag{3.5.6}$$

For all objects $X, x_{\mathcal{A}}, x_{\mathcal{B}}$ and $Y, y_{\mathcal{A}}, y_{\mathcal{B}}$. This follows from the fact that $\prod$ and $\sum$ preserve homotopical structure. 3.5.6 gives us an equality principle for arrows in this category. The equality principle says that to prove two arrows (f, f_0, f_1) and (g, g_0, g_1) equal it suffices to give a proof that f and g are equal. And thus we have proven 3.5.3 using 3.5.4.

We must now prove that the arrows form a set:

$$isSet\ (Arrow\ \overline{X}\ \overline{Y})$$

Since pairs preserve homotopical structure this reduces to the following two obligations: The first one is:

$$isSet\ (\mathbb{C}.Arrow\ X\ Y)$$

which holds. The other one is:

$$\prod_{f:Arrow\ X\ Y} isSet\ (y_{\mathcal{A}} \lll f \equiv x_{\mathcal{A}} \times y_{\mathcal{B}} \lll f \equiv x_{\mathcal{B}})$$

We get this from 3.5.6 and the fact that homotopical structure is cumulative.

That concludes the proof that this is a valid pre-category.

Univalence

To prove that this is a proper category we must additionally prove the univalence criterion. That is, for any two objects $\mathcal{X} = (X, x_{\mathcal{A}}, x_{\mathcal{B}})$ and $\mathcal{Y} = Y, y_{\mathcal{A}}, y_{\mathcal{B}}$ I will show:

$$(\mathcal{X} \equiv \mathcal{Y}) \cong (\mathcal{X} \cong \mathcal{Y}) \tag{3.5.7}$$

I do this by showing that the following sequence of types are isomorphic.

The first type is:

$$(X, x_{\mathcal{A}}, x_{\mathcal{B}}) \equiv (Y, y_{\mathcal{A}}, y_{\mathcal{B}}) \tag{3.5.8}$$

The next types will be the triple:

$$\begin{aligned} p : & X \equiv Y \\ & Path\ (\lambda\ \ i \to Arrow\ (p\ i)\ \mathcal{A})\ x_{\mathcal{A}}\ y_{\mathcal{A}} \\ & Path\ (\lambda\ \ i \to Arrow\ (p\ i)\ \mathcal{B})\ x_{\mathcal{B}}\ y_{\mathcal{B}} \end{aligned} \tag{3.5.9}$$

The next type is very similar, but instead of a path we will have an isomorphism and create a path from this:

$$\begin{aligned} iso : & X \cong Y \\ & Path\ (\lambda\ \ i \to Arrow\ (\tilde{p}\ i)\ \mathcal{A})\ x_{\mathcal{A}}\ y_{\mathcal{A}} \\ & Path\ (\lambda\ \ i \to Arrow\ (\tilde{p}\ i)\ \mathcal{B})\ x_{\mathcal{B}}\ y_{\mathcal{B}} \end{aligned} \tag{3.5.10}$$

where $\tilde{p} \triangleq isoToId\ iso : X \equiv Y$.

Finally we have the type:

$$(X, x_{\mathcal{A}}, x_{\mathcal{B}}) \cong (Y, y_{\mathcal{A}}, y_{\mathcal{B}}) \tag{3.5.11}$$

The proof is a chain of isomorphisms between the types 3.5.8, 3.5.9, 3.5.10 and 3.5.11. I will highlight the most interesting bits of this proof. For the full proof see the implementation in the same module:

`Cat.Categories.Span` ↗

Proposition 3.5.8 is isomorphic to 3.5.9: This is just an application of the fact that a path between two pairs a_0, a_1 and b_0, b_1 corresponds to a pair of paths between a_0, b_0 and a_1, b_1.

Proposition 3.5.9 is isomorphic to 3.5.10: This proof of this has been omitted but can be found in the module:

`Cat.Categories.Span` ↗

Proposition 3.5.10 is isomorphic to 3.5.11: For this I will show two corollaries of 3.3.2: For an isomorphism $(\iota, \iota^{-1}, inv) : A \cong B$, arrows $f : Arrow\ A\ X$, $g : Arrow\ B\ X$ and a heterogeneous path between them, $q : Path\ (\lambda\ i \to p_{dom}\ i)\ f\ g$, where $p_{dom} : Arrow\ A\ X \equiv Arrow\ B\ X$ is a path induced by *iso*, we have the following two results

$$f \equiv g \lll \iota \tag{3.5.12}$$

$$g \equiv f \lll \iota^{-1} \tag{3.5.13}$$

The proof is omitted but can be found in the module:

`Cat.Category` ↗

Now we can prove the equivalence in the following way: Given $(f, f^{-1}, inv_f) : X \cong Y$ and two heterogeneous paths

$$p_{\mathcal{A}} : Path\ (\lambda\ i \to p_{dom}\ i)\ x_{\mathcal{A}}\ y_{\mathcal{A}}$$
$$q_{\mathcal{B}} : Path\ (\lambda\ i \to p_{dom}\ i)\ x_{\mathcal{B}}\ y_{\mathcal{B}}$$

all as in 3.5.10. I use p_{dom} here again to mean the path induced by the isomorphism (f, f^{-1}, inv_f). We must now construct an isomorphism $(X, x_{\mathcal{A}}, x_{\mathcal{B}}) \cong (Y, y_{\mathcal{A}}, y_{\mathcal{B}})$ as in 3.5.11. That is, an isomorphism in the present category. I remind the reader that such a gadget is a triple. The first component shall be:

$$f : Arrow\ X\ Y \tag{3.5.14}$$

To show that this choice fits the bill, I must verify that it satisfies 3.5.2, which in this case becomes:

$$(y_{\mathcal{A}} \lll f \equiv x_{\mathcal{A}}) \times (y_{\mathcal{B}} \lll f \equiv x_{\mathcal{B}}) \tag{3.5.15}$$

which, since f is an isomorphism and $p_{\mathcal{A}}$ (resp. $p_{\mathcal{B}}$) is a path varying according to a path constructed from this isomorphism, this is exactly what 3.5.12 gives us. The other direction is quite analogous. We choose f^{-1} as the morphism and prove that it satisfies 3.5.2 with 3.5.13.

We must now show that this choice of arrows indeed form an isomorphism. Our equality principle for arrows in this category (3.5.6) gives us that it suffices to show that f and f^{-1} are inverses, but this is of course just inv_f.

This concludes the first direction of the isomorphism that we are constructing. For the other direction we are given the isomorphism:

$$(f, f^{-1}, inv_f, inv_{f^{-1}}) : (X, x_{\mathcal{A}}, x_{\mathcal{B}}) \cong (Y, y_{\mathcal{A}}, y_{\mathcal{B}})$$

Projecting out the first component gives us the isomorphism

$$(fst\ f, fst\ f^{-1}, cong\ fst\ inv_f, cong\ fst\ inv_{f^{-1}}) : X \cong Y$$

This gives rise to the following paths:

$$\begin{aligned} \tilde{p} &: X \equiv Y \\ \tilde{p}_{\mathcal{A}} &: Arrow\ X\ \mathcal{A} \equiv Arrow\ Y\ \mathcal{A} \\ \tilde{p}_{\mathcal{B}} &: Arrow\ X\ \mathcal{B} \equiv Arrow\ Y\ \mathcal{B} \end{aligned} \tag{3.5.16}$$

It then remains to construct the two paths:

$$\begin{aligned} &Path\ (\lambda\ \ i \to \tilde{p}_{\mathcal{A}}\ i)\ x_{\mathcal{A}}\ y_{\mathcal{A}} \\ &Path\ (\lambda\ \ i \to \tilde{p}_{\mathcal{B}}\ i)\ x_{\mathcal{B}}\ y_{\mathcal{B}} \end{aligned} \tag{3.5.17}$$

This is achieved with the following lemma:

$$\prod_{a:A} \prod_{b:B} \prod_{q:A\equiv B} coe\ q\ a \equiv b \to Path\ (\lambda\ \ i \to q\ i)\ a\ b \tag{3.5.18}$$

which is used without proof. See the implementation for the details.

3.5.17 is then proven with the propositions:

$$\begin{aligned} coe\ \tilde{p}_{\mathcal{A}}\ x_{\mathcal{A}} &\equiv y_{\mathcal{A}} \\ coe\ \tilde{p}_{\mathcal{B}}\ x_{\mathcal{B}} &\equiv y_{\mathcal{B}} \end{aligned} \tag{3.5.19}$$

The proof of the first one is:

$$\begin{aligned} coe\ \tilde{p}_{\mathcal{A}}\ x_{\mathcal{A}} \equiv x_{\mathcal{A}} \lll fst\ f^{-1} \qquad & \text{3.3.1 and the isomorphism } (f, f^{-1}, inv_f) \\ \equiv y_{\mathcal{A}} \qquad & \text{3.5.2 for } f^{-1} \end{aligned}$$

We have now constructed the maps between 3.5.8 and 3.5.9. It remains to show that they are inverses of each other. To cut a long story short, the proof uses the fact that isomorphism-of is a proposition and that arrows (in both categories) are sets. The reader is referred to the implementation for the full gory details.

3.5.3 To have products is a property of a category

Now that we have constructed the span category I will demonstrate how to use this to prove that products are propositions. On the face of it this may seem surprising. Products look like they are a structure on categories. After all it consist of a select object and two arrows. If formulated in set theory this would be the case but in the present setting univalence of categories give us that products are properties. I will show this by showing that terminal objects in the span category are equivalent to products:

$$\mathit{Terminal} \simeq \mathit{Product}\ \mathbb{C}\ \mathcal{A}\ \mathcal{B} \tag{3.5.20}$$

and as always we do this by constructing an isomorphism: In the direction

$$\mathit{Terminal} \to \mathit{Product}\ \mathbb{C}\ \mathcal{A}\ \mathcal{B}$$

we are given a terminal object $X, x_{\mathcal{A}}, x_{\mathcal{B}}$. X will be the product-object and $x_{\mathcal{A}}, x_{\mathcal{B}}$ will be the product arrows, so it just remains to verify that this is indeed a product. That is, for an object Y and two arrows $y_{\mathcal{A}} : \mathit{Arrow}\ Y\ \mathcal{A}$, $y_{\mathcal{B}}\ \mathit{Arrow}\ Y\ \mathcal{B}$ we must find a unique arrow $f : \mathit{Arrow}\ Y\ X$ satisfying:

$$(x_{\mathcal{A}} \lll f \equiv y_{\mathcal{A}}) \times (x_{\mathcal{B}} \lll f \equiv y_{\mathcal{B}}) \tag{3.5.21}$$

Since $X, x_{\mathcal{A}}, x_{\mathcal{B}}$ is a terminal object, there is a *unique* arrow from this object to any other object. In particular we have $Y, y_{\mathcal{A}}, y_{\mathcal{B}}$. The arrow we will play the role of f and it immediately satisfies 3.5.21. Any other arrow satisfying these conditions will be equal since f is unique.

For the other direction we are now given a product $X, x_{\mathcal{A}}, x_{\mathcal{B}}$. Again this will be the terminal object. Now it remains that for any other object there is a unique arrow from that object into $X, x_{\mathcal{A}}, x_{\mathcal{B}}$. Let $Y, y_{\mathcal{A}}, y_{\mathcal{B}}$ be another object. As the arrow $\mathit{Arrow}\ Y\ X$ we choose the product-arrow $y_{\mathcal{A}} \times y_{\mathcal{B}}$. Since this is a product-arrow it satisfies 3.5.21. Let us name the witness to this $\phi_{y_{\mathcal{A}} \times y_{\mathcal{B}}}$. We have picked as our center of contraction $y_{\mathcal{A}} \times y_{\mathcal{B}}, \phi_{y_{\mathcal{A}} \times y_{\mathcal{B}}}$ we must now show that it is contractible. Let $f : \mathit{Arrow}\ X\ Y$ and ϕ_f be given (here ϕ_f is the proof that f satisfies 3.5.21). The proof will be a pair of proofs:

$$p : \mathit{Path}\ (\lambda\ i \to \mathit{Arrow}\ X\ Y) \quad f \quad y_{\mathcal{A}} \times y_{\mathcal{B}} \tag{3.5.22}$$

$$\mathit{Path}\ (\lambda\ i \to \Phi\ (p\ i)) \quad \phi_f \quad \phi_{y_{\mathcal{A}} \times y_{\mathcal{B}}} \tag{3.5.23}$$

Here Φ is given as:

$$\prod_{f : \mathit{Arrow}\ Y\ X} (x_{\mathcal{A}} \lll f \equiv y_{\mathcal{A}}) \times (x_{\mathcal{B}} \lll f \equiv y_{\mathcal{B}})$$

We can construct p from the universal property of $y_{\mathcal{A}} \times y_{\mathcal{B}}$. For the latter we use the same trick we did in 3.4.4 and prove this more general result:

$$\prod_{f : \mathit{Arrow}\ Y\ X} \mathit{isProp}\ ((x_{\mathcal{A}} \lll f \equiv y_{\mathcal{A}}) \times (x_{\mathcal{B}} \lll f \equiv y_{\mathcal{B}}))$$

Which follows from arrows being sets and pairs preserving such. Thus we can close the final proof with an application of *lemPropF*.

This concludes the proof $\mathit{Terminal} \simeq \mathit{Product}\ \mathbb{C}\ \mathcal{A}\ \mathcal{B}$ and since we have that equivalences preserve homotopic levels along with 3.1.15 we get our final result. That is, in any category $\mathbb{C}$ we have:

$$\prod_{A,B : \mathit{Object}} \mathit{isProp}\ (\mathit{Product}\ \mathbb{C}\ A\ B) \tag{3.5.24}$$

3.6 Functors and natural transformations

For the sake of completeness I will briefly mention the definition of functors and natural transformations. Please refer to the implementation for the full details.

3.6.1 Functors

Given two categories $\mathbb{C}$ and $\mathbb{D}$ a functor consists of the following data:

$$\begin{aligned} \mathcal{F} &: \mathbb{C}.Object \to \mathbb{D}.Object \\ fmap &: \mathbb{C}.Arrow\ A\ B \to \mathbb{D}.Arrow\ (\mathcal{F}\ A)\ (\mathcal{F}\ B) \end{aligned}$$

and the following laws:

$$\begin{aligned} fmap\ \mathbb{C}.identity &\equiv \mathbb{D}.identity \\ fmap\ (g\ \mathbb{C}.\lll f) &\equiv fmap\ g\ \mathbb{D}.\lll fmap\ f \end{aligned}$$

The implementation can be found here:

`Cat.Category.Functor`

3.6.2 Natural Transformation

Given two functors between categories $\mathbb{C}$ and $\mathbb{D}$. Name them $\widehat{\mathcal{F}}$ and $\widehat{\mathcal{G}}$. A natural transformation is a family of arrows:

$$\prod_{C:\mathbb{C}.Object} \mathbb{D}.Arrow\ (\mathcal{F}\ C)\ (\mathcal{G}\ C)$$

This family of arrows can be seen as the data. If θ is a natural transformation $\theta\ C$ will be called the component (of θ) at C. The laws of this family of morphism is the naturality condition:

$$\prod_{f:\mathbb{C}.Arrow\ A\ B} (\theta\ B)\ \mathbb{D}.\lll (\widehat{\mathcal{F}}.fmap\ f) \equiv (\widehat{\mathcal{G}}.fmap\ f)\ \mathbb{D}.\lll (\theta\ A)$$

The implementation can be found here:

`Cat.Category.NaturalTransformation`

3.7 Monads

In this section I present two formulations of monads. The two representations are referred to as the monoidal- and Kleisli- representation respectively or simply monoidal monads and Kleisli monads. We then show that the two formulations are equivalent, which due to univalence gives us a path between the two types.

Let a category $\mathbb{C}$ be given. In the remainder of this sections all objects and arrows will implicitly refer to objects and arrows in this category. I will also use the notation $\widehat{\mathcal{R}}$ to refer to an

endofunctor on this category. Its map on objects will be denoted $\mathcal{R}$ and its map on arrows will be denoted *fmap*. Likewise I will use the notation $\widehat{pure}$ to refer to a natural transformation and its component at a given (implicit) object will be denoted *pure*. This is the same notation as in §3.6.2.

3.7.1 Monoidal formulation

The monoidal formulation of monads consists of the following data:

$$\begin{aligned} \widehat{\mathcal{R}} &: Functor\ \mathbb{C}\ \mathbb{C} \\ \widehat{pure} &: NT\ \widehat{identity}\ \widehat{\mathcal{R}} \\ \widehat{join} &: NT\ (\widehat{\mathcal{R}} \oplus \widehat{\mathcal{R}})\ \widehat{\mathcal{R}} \end{aligned} \tag{3.7.1}$$

Here NT denotes natural transformations and $\oplus$ means functor composition.

Note that *fmap* is $\widehat{\mathcal{R}}$'s map on arrows. This data must satisfy the following laws.

$$join \lll fmap\ join \equiv join \lll join \tag{3.7.2}$$

$$join \lll pure \equiv identity \tag{3.7.3}$$

$$join \lll fmap\ pure \equiv identity \tag{3.7.4}$$

The implicit arguments to the arrows above have been left out and the objects they range over are universally quantified.

3.7.2 Kleisli formulation

The Kleisli-formulation consists of the following data:

$$\begin{aligned} \mathcal{R} &: Object \to Object \\ pure &: Arrow\ X\ (\mathcal{R}\ X) \\ bind &: Arrow\ X\ (\mathcal{R}\ Y) \to Arrow\ (\mathcal{R}\ X)\ (\mathcal{R}\ Y) \end{aligned} \tag{3.7.5}$$

The objects X and Y are implicitly universally quantified. With this data we can construct the *Kleisli arrow*:

$$\begin{aligned} \mapsto &: Arrow\ A\ (\mathcal{R}\ B) \to Arrow\ B\ (\mathcal{R}\ C) \to Arrow\ A\ (\mathcal{R}\ C) \\ f \mapsto g &\triangleq f \ggg (bind\ g) \end{aligned}$$

It is interesting to note here that this formulation does not mention functors nor natural transformations. All we have here is a regular map on objects and a pair of arrows. This data must satisfy:

$$bind\ pure \equiv identity_{\mathcal{R}\ X} \tag{3.7.6}$$

$$pure \mapsto f \equiv f \tag{3.7.7}$$

$$(bind\ f) \ggg (bind\ g) \equiv bind\ (f \mapsto g) \tag{3.7.8}$$

Here likewise the arrows $f : Arrow\ X\ (\mathcal{R}\ Y)$ and $g : Arrow\ Y\ (\mathcal{R}\ Z)$ are universally quantified as well as the objects they range over. The third law is stated in terms of reverse function composition to mirror the way in which it interacts with the Kleisli arrow.

3.7.3 Equivalence of formulations

Both formulations mention a map called *pure*. This is of course no coincidence as that arrow in the Kleisli formulation shall correspond exactly to the map on arrows for the natural transformation in the monoidal formulation.

In the monoidal formulation we can define *bind*:

$$bind\ f \triangleq join \lll fmap\ f \tag{3.7.9}$$

and likewise in the Kleisli formulation we can define *join*:

$$join \triangleq bind\ identity \tag{3.7.10}$$

It now remains to show that this construction indeed gives rise to a monad. This will be done in two steps. First we will assume that we have a monad in the monoidal form; $(\widehat{\mathcal{R}}, pure, join)$ and then show that $(\mathcal{R}, pure, bind)$ is indeed a monad in the Kleisli form. In the second part we will show the other direction.

Monoidal to Kleisli

Let $(\widehat{\mathcal{R}}, \widehat{pure}, \widehat{join})$ be given as in 3.7.1 satisfying the laws 3.7.2, 3.7.3 and 3.7.4. For the data of the Kleisli formulation we pick:

$$\begin{aligned} \mathcal{R} &\triangleq \mathcal{R} \\ pure &\triangleq pure \\ bind\ f &\triangleq join \lll fmap\ f \end{aligned} \tag{3.7.11}$$

Again $\mathcal{R}$ is the object map of the endo-functor $\widehat{\mathcal{R}}$, *pure* and *join* are the arrows from the natural transformations $\widehat{pure}$ and $\widehat{join}$ respectively and *fmap* is the map on arrows of the endofunctor $\widehat{\mathcal{R}}$. It now just remains to verify the laws 3.7.6, 3.7.7 and 3.7.8. For 3.7.6:

$$\begin{aligned} bind\ pure &= join \lll (fmap\ pure) \\ &\equiv identity && \text{By 3.7.4} \end{aligned}$$

For 3.7.7:

$$\begin{aligned} pure \mapsto f &= pure \ggg bind\ f \\ &= bind\ f \lll pure \\ &= join \lll fmap\ f \lll pure \\ &\equiv join \lll pure \lll f && pure \text{ is a natural transformation} \\ &\equiv identity \lll f && \text{By 3.7.3} \\ &\equiv f && \text{Left identity} \end{aligned}$$

For 3.7.8:

$$
\begin{aligned}
\mathit{bind}\ g \ggg \mathit{bind}\ f &= \mathit{bind}\ f \lll \mathit{bind}\ g \\
&= \mathit{join} \lll \mathit{fmap}\ g \lll \mathit{join} \lll \mathit{fmap}\ f \\
&\equiv \mathit{join} \lll \mathit{join} \lll (\mathit{fmap} \circ \mathit{fmap})\ f \lll \mathit{fmap}\ g && \mathit{join} \text{ is a natural transformation} \\
&\equiv \mathit{join} \lll \mathit{fmap}\ \mathit{join} \lll (\mathit{fmap} \circ \mathit{fmap})\ f \lll \mathit{fmap}\ g && \text{By 3.7.2} \\
&= \mathit{join} \lll \mathit{fmap}\ \mathit{join} \lll \mathit{fmap}\ (\mathit{fmap}\ f) \lll \mathit{fmap}\ g \\
&\equiv \mathit{join} \lll \mathit{fmap}\ (\mathit{join} \lll \mathit{fmap}\ f \lll g) && \text{Distributive law for functors} \\
&= \mathit{join} \lll \mathit{fmap}\ (\mathit{join} \lll \mathit{fmap}\ f \lll g) \\
&= \mathit{bind}\ (\mathit{bind}\ f \lll g) \\
&= \mathit{bind}\ (g \ggg \mathit{bind}\ f) \\
&= \mathit{bind}\ (g \mapsto f)
\end{aligned}
$$

The construction can be found in the module:

`Cat.Category.Monad.Monoidal`

Kleisli to Monoidal

For the other direction we are given $(\mathcal{R}, \mathit{pure}, \mathit{bind})$ as in 3.7.5 satisfying the laws 3.7.6, 3.7.7 and 3.7.8. For the data of the monoidal formulation we pick:

$$
\begin{aligned}
\widehat{\mathcal{R}} &\triangleq (\mathcal{R}, \mathit{bind}\ (\mathit{pure} \lll f)) \\
\mathit{pure} &\triangleq \mathit{pure} \\
\mathit{join} &\triangleq \mathit{bind}\ \mathit{identity}
\end{aligned}
\tag{3.7.12}
$$

We must now not only show the monad laws given for the monoidal formulation (3.7.2, 3.7.3 and 3.7.4), we must also verify that $\widehat{\mathcal{R}}$ is a functor and that *pure* and *join* are natural transformations. I will ommit this here. Instead we shall see how these two mappings are indeed inverses. The full construction can be found in the module:

`Cat.Category.Monad.Kleisli`

Equivalence

To prove that the two formulations are equivalent we must demonstrate that the two mappings sketched above are indeed inverses of each other. To recap, these maps are:

$$
\begin{aligned}
&\mathit{toKleisli} : \mathit{Kleisli} \to \mathit{Monoidal} \\
&\mathit{toKleisli} \triangleq \lambda\ (\mathcal{R}, \mathit{pure}, \mathit{bind}) \to (\widehat{\mathcal{R}}, \mathit{pure}, \mathit{bind}\ \mathit{identity})
\end{aligned}
$$

where $\widehat{\mathcal{R}} \triangleq (\mathcal{R}, \mathit{bind}\ (\mathit{pure} \lll f))$. The proof that this is indeed a functor is left implicit as well as the monad laws. Likewise the proof that *pure* and *bind identity* are natural transformations are left implicit. The inverse map will be:

$$
\begin{aligned}
&\mathit{toMonoidal} : \mathit{Monoidal} \to \mathit{Kleisli} \\
&\mathit{toMonoidal} \triangleq \lambda\ (\widehat{\mathcal{R}}, \widehat{\mathit{pure}}, \widehat{\mathit{join}}) \to (\mathcal{R}, \mathit{pure}, \mathit{bind})
\end{aligned}
$$

Where $bind\ f \triangleq join \lll fmap\ f$. Again the monad laws are left implicit. Now we must show:

$$toKleisli \circ toMonoidal \equiv identity \tag{3.7.13}$$

$$toMonoidal \circ toKleisli \equiv identity \tag{3.7.14}$$

For 3.7.13 let $(\mathcal{R}, pure, bind)$ be a monad in the Kleisli form. Since being-a-monad is a proposition[2] we get an equality-principle for kleisli-monads that say that to equate two such monads it suffices to equate their data part. It thus suffices to equate the data parts of 3.7.13. Such a proof is a triple equating the three projections of 3.7.5. The first two hold definitionally – essentially one just wraps and unwraps the morphism in a functor. For the last equation a little more work is required:

$$\begin{aligned}
join \lll fmap\ f &= fmap\ f \ggg join \\
&= bind\ (f \ggg pure) \ggg bind\ identity && \text{By definition of } fmap \text{ and } join \\
&\equiv bind\ (f \ggg pure \mapsto identity) && \text{By 3.7.8} \\
&\equiv bind\ (f \ggg identity) && \text{By 3.7.7} \\
&= bind\ f
\end{aligned}$$

For the other direction (3.7.14) we are given a monad in the monoidal form; $(\widehat{\mathcal{R}}, \widehat{pure}, \widehat{join})$. The various equality-principles again give us that it is sufficient to equate the data-part of the above. That is, we only need to verify that the following pieces of data: $\mathcal{R}$, *fmap*, *pure* and *join* get mapped correctly. To see the full details check the implementation in the module:

`Cat.Category.Monad`

[2]The proof was omitted here but can be found in the implementation.

Chapter 4

Perspectives

4.1 Discussion

In the previous chapter the practical aspects of proving things in Cubical Agda were highlighted. I also demonstrated the usefulness of separating "laws" from "data". One of the reasons for this is that dependencies within types can lead to very complicated goals. One technique for alleviating this was to prove that certain types are mere propositions.

4.1.1 Computational properties

The new contribution of cubical Agda is that it has a constructive proof of functional extensionality and univalence. This means in particular that the type checker can reduce terms defined with these theorems. One interesting result of this development is how much this influenced the development. In particular having a functional extensionality that "computes" should simplify some proofs.

I have tested this by using a feature of Agda where one can mark certain bindings as being *abstract*. This means that the type-checker will not try to reduce that term further during type checking. I tried making univalence and functional extensionality abstract. It turns out that the conversion behaviour of univalence is not used anywhere. For functional extensionality there are two places in the whole solution where the reduction behaviour is used to simplify some proofs. This is in showing that the maps between the two formulations of monads are inverses. See the notes in this module:

`Cat.Category.Monad.Voevodsky`

I will not reproduce it in full here as the type is quite involved. In stead I have put this in a source listing in A. The method used to find in what places the computational behaviour of these proofs are needed has the caveat of only working for places that directly or transitively uses these two proofs. Fortunately though the code is structured in such a way that this is the case. In conclusion the way I have structured these proofs means that the computational behaviour of functional extensionality and univalence has not been so relevant.

Barring this the computational behaviour of paths can still be useful. E.g. if a programmer wants to reuse functions that operate on a monoidal monads to work with a monad in the Kleisli form that the programmer has specified. To make this idea concrete, say we are given some function $f : \mathit{Kleisli} \to T$, having a path between $p : \mathit{Monoidal} \equiv \mathit{Kleisli}$ induces a map $\mathit{coe}\, p : \mathit{Monoidal} \to \mathit{Kleisli}$. We can compose f with this map to get $f \circ \mathit{coe}\, p : \mathit{Monoidal} \to T$. Of course, since that map was constructed with an isomorphism, these maps already exist and could be used directly. So this is arguably only interesting when one also wants to prove properties of applying such functions.

4.1.2 Reusability of proofs

The previous example illustrate how univalence unifies two otherwise disparate areas: The category-theoretic study of monads; and monads as in functional programming. Univalence thus allows one to reuse proofs. You could say that univalence gives the developer two proofs for the price of one. As an illustration of this I proved that monads are groupoids. I initially proved this for the Kleisli formulation[1]. Since the two formulations are equal under univalence, substitution directly gives us that this also holds for the monoidal formulation. This of course generalizes to any family $P : \mathcal{U} \to \mathcal{U}$ where P is inhabited at either formulation (i.e. either P *Monoidal* or P *Kleisli* holds).

The introduction (section §1.3) mentioned that a typical way of getting access to functional extensionality is to work with setoids. Nowhere in this formalization has this been necessary, *Path* has been used globally in the project for propositional equality. One interesting place where this becomes apparent is in interfacing with the Agda standard library. Multiple definitions in the Agda standard library have been designed with the setoid-interpretation in mind. E.g. the notion of *unique existential* is indexed by a relation that should play the role of propositional equality. Equivalence relations are likewise indexed, not only by the actual equivalence relation but also by another relation that serve as propositional equality.

In the formalization at present a significant amount of energy has been put towards proving things that would not have been needed in classical Agda. The proofs that some given type is a proposition were provided as a strategy to simplify some otherwise very complicated proofs (e.g. 3.1.14 and 3.5.5). Often these proofs would not be this complicated. If the J-rule holds definitionally the proof-assistant can help simplify these goals considerably. The lack of the J-rule has a significant impact on the complexity of these kinds of proofs.

4.1.3 Motifs

An oft-used technique in this development is using based path induction to prove certain properties. One particular challenge that arises when doing so is that Agda is not able to automatically infer the family that one wants to do induction over. For instance in the proof $\mathit{sym}\ (\mathit{sym}\ p) \equiv p$ from 2.3.2 the family that we chose to do induction over was $D\ b'\ p' \triangleq \mathit{sym}\ (\mathit{sym}\ p') \equiv p'$. However, if one interactively tries to give this hole, all the information that Agda can provide is that one must provide an element of $\mathcal{U}$. Agda could be more helpful in this context, perhaps even infer this family in some situations. In this very simple example this is of course not a big problem, but there are examples in the source code where this gets more involved.

[1] Actually doing this directly turned out to be tricky as well, so I defined an equivalent formulation which was not formulated with a record, but purely with $\sum$-types.

4.2 Future work

4.2.1 Compiling Cubical Agda

Compilation of program written in Cubical Agda is currently not supported. One issue here is that the backends does not provide an implementation for the cubical primitives (such as the path-type). This means that even though the path-type gives us a computational interpretation of functional extensionality, univalence, transport, etc., we do not have a way of actually using this to compile our programs that use these primitives. It would be interesting to see practical applications of this.

4.2.2 Proving laws of programs

Another interesting thing would be to use the Kleisli formulation of monads to prove properties of functional programs. The existence of univalence will make it possible to re-use proofs stated in terms of the monoidal formulation in this setting.

4.2.3 Initiality conjecture

A fellow student at Chalmers, Andreas Källberg, is currently working on proving the initiality conjecture. He will be using this library to do so.

Chapter 5

Conclusion

This thesis highlighted some issues with the standard inductive definition of propositional equality used in Agda. Functional extensionality and univalence are examples of two propositions not admissible in Intensional Type Theory (ITT). This has a big impact on what is provable and the reusability of proofs. This issue is overcome with an extension to Agda's type system called Cubical Agda. With Cubical Agda both functional extensionality and univalence are admissible. Cubical Agda is more expressive, but there are certain issues that arise that are not present in standard Agda. For one thing Agda enjoys Uniqueness of Identity Proofs (UIP) though a flag exists to turn this off. This feature is not present in Cubical Agda. Rather than having unique identity proofs cubical Agda gives rise to a hierarchy of types with increasing *homotopical structure*. It turns out to be useful to build the formalization with this hierarchy in mind as it can simplify proofs considerably. Another issue one must overcome in Cubical Agda is when a type has a field whose type depends on a previous field. In this case paths between such types will be heterogeneous paths. In practice it turns out to be considerably more difficult to work with heterogeneous paths than with homogeneous paths. This thesis demonstrated the application of some techniques to overcome these difficulties, such as based path induction.

This thesis formalizes some of the core concepts from category theory including: categories, functors, products, exponentials, Cartesian closed categories, natural transformations, the yoneda embedding, monads and more. Category theory is an interesting case study for the application of cubical Agda for two reasons in particular. One reason is because category theory is the study of abstract algebra of functions, meaning that functional extensionality is particularly relevant. Another reason is that in category theory it is commonplace to identify isomorphic structures. Univalence allows for making this notion precise. This thesis also demonstrated another technique that is common in category theory; namely to define categories to prove properties of other structures. Specifically a category was defined to demonstrate that any two product objects in a category are isomorphic. Furthermore the thesis showed two formulations of monads and proved that they indeed are equivalent: Namely monads in the monoidal- and Kleisli- form. The monoidal formulation is more typical to category theoretic formulations and the Kleisli formulation will be more familiar to functional programmers. It would have been very difficult to make a similar proof with setoids and the proof would be very difficult to read. In the formulation we also saw how paths can be used to extract functions. A path between two types induce an isomorphism between the two types. This e.g. permits developers to write a monad instance for a given type using the Kleisli formulation. By transporting along the path between the monoidal- and Kleisli- formulation one can reuse all the operations and results shown for monoidal- monads in the context of kleisli monads.

Bibliography

[1] S. Awodey. *Category Theory*. Oxford Logic Guides. Ebsco Publishing, 2006.

[2] Thierry Coquand and Nils Anders Danielsson. Isomorphism is equality. *Indagationes Mathematicae*, 24(4):1105–1120, 2013.

[3] Martin Hofmann. Extensional concepts in intensional type theory. 1995.

[4] Simon Huber. Cubical intepretations of type theory. 2016.

[5] The Univalent Foundations Program. *Homotopy Type Theory: Univalent Foundations of Mathematics*. `https://homotopytypetheory.org/book`, Institute for Advanced Study, 2013.

[6] Andrea Vezzosi. Cubical extension to agda. `https://github.com/agda/agda`, 2017.

[7] Andrea Vezzosi. Cubical type theory demo. `https://github.com/Saizan/cubical-demo`, 2017.

Appendices

Appendix A

Non-reducing functional extensionality

In two places in my formalization was the computational behaviours of functional extensionality used. The reduction behaviour can be disabled by marking functional extensionality as abstract. Below the fully normalized goal and context with functional extensionality marked abstract has been shown. The excerpts are from the module

Cat.Category.Monad.Voevodsky

where this is also written as a comment next to the proofs. When functional extensionality is not abstract the goal and current value are the same. It is of course necessary to show the fully normalized goal and context otherwise the reduction behaviours is not forced.

First goal

Goal:

```
PathP (λ _ → §2-3.§2 omap (λ {z} → pure))
(§2-fromMonad
 (.Cat.Category.Monad.toKleisli ℂ
  (.Cat.Category.Monad.toMonoidal ℂ (§2-3.§2.toMonad m))))
(§2-fromMonad (§2-3.§2.toMonad m))
```

Have:

```
PathP
(λ i →
   §2-3.§2 K.IsMonad.omap
   (K.RawMonad.pure
    (K.Monad.raw
    (funExt (λ m₁ → K.Monad≡ (.Cat.Category.Monad.toKleisliRawEq ℂ m₁))
      i (§2-3.§2.toMonad m)))))
(§2-fromMonad
```

```
 (.Cat.Category.Monad.toKleisli ℂ
  (.Cat.Category.Monad.toMonoidal ℂ (§2-3.§2.toMonad m))))
(§2-fromMonad (§2-3.§2.toMonad m))
```

Second goal

Goal:

```
PathP (λ _ → §2-3.§1 omap (λ {X} → pure))
(§1-fromMonad
 (.Cat.Category.Monad.toMonoidal ℂ
  (.Cat.Category.Monad.toKleisli ℂ (§2-3.§1.toMonad m))))
(§1-fromMonad (§2-3.§1.toMonad m))
```

Have:

```
PathP
(λ i →
   §2-3.§1
   (RawFunctor.omap
    (Functor.raw
     (M.RawMonad.R
      (M.Monad.raw
       (funExt
        (λ m₁ → M.Monad≡ (.Cat.Category.Monad.toMonoidalRawEq ℂ m₁)) i
        (§2-3.§1.toMonad m))))))
   (λ {X} →
      fst
      (M.RawMonad.pureNT
       (M.Monad.raw
        (funExt
        (λ m₁ → M.Monad≡ (.Cat.Category.Monad.toMonoidalRawEq ℂ m₁)) i
         (§2-3.§1.toMonad m))))
      X))
(§1-fromMonad
 (.Cat.Category.Monad.toMonoidal ℂ
  (.Cat.Category.Monad.toKleisli ℂ (§2-3.§1.toMonad m))))
(§1-fromMonad (§2-3.§1.toMonad m))
```

Index