KU LEUVEN

FACULTEIT WETENSCHAPPEN

Towards a Directed Homotopy Type Theory based on 4 Kinds of Variance

Andreas NUYTS

Promotor: Prof. dr. ir. F. Piessens

Begeleiders:

Dr. D. Devriese, J. Cockx

DistriNet,
Departement
Computerwetenschappen,
KU Leuven

Proefschrift ingediend tot het
behalen van de graad van
Master in de Wiskunde

Academiejaar 2014-2015

Towards a Directed Homotopy Type Theory based on 4 Kinds of Variance

Andreas Nuyts

Promotor: Prof. dr. ir. Frank Piessens

Begeleiders: Dr. Dominique Devriese, Jesper Cockx

Contents

Contents iii

Voorwoord vii

Vulgariserende samenvatting ix

English abstract xi

Nederlandstalige abstract xiii

1 Introduction 1
- 1.1 Dependent type theory . . . 1
- 1.2 Homotopy type theory . . . 5
- 1.3 Directed type theory . . . 6
  - 1.3.1 Two-dimensional directed type theory . . . 6
  - 1.3.2 Other related work . . . 8
  - 1.3.3 Directed homotopy type theory . . . 9
  - 1.3.4 Contributions . . . 11

2 Dependent type theory, symmetric and directed 13
- 2.1 Judgements and contexts . . . 13
- 2.2 Structural inference rules . . . 18
- 2.3 Universe types . . . 20
- 2.4 Non-dependent function types . . . 21
- 2.5 Dependent function types . . . 25
- 2.6 Intermezzo: equivalences and function extensionality . . . 27
- 2.7 Inductive types and the coproduct . . . 28
- 2.8 Lists . . . 34
- 2.9 The naturals . . . 36
- 2.10 The family of finite sets . . . 38
- 2.11 The product . . . 40
- 2.12 Opposite, core and localization . . . 44
  - 2.12.1 The opposite of a type . . . 44
  - 2.12.2 The core of a type . . . 47
  - 2.12.3 Bridged groupoids . . . 49
  - 2.12.4 The localization of a type . . . 50
  - 2.12.5 Bridgeless groupoids . . . 52
- 2.13 Dependent pair types . . . 53

2.14 Inductive type families: some examples . . . 57
2.15 The identity type . . . 64
2.16 The morphism type . . . 70
2.17 W-types . . . 74
2.18 Higher inductive types . . . 75

3 Homotopy type theory, symmetric and directed 77
3.1 Types are higher categories . . . 77
3.2 Functions are functors . . . 82
3.3 Equivalences, isomorphisms and transport . . . 86
3.4 Type families are fibrations . . . 91
3.4.1 In symmetric HoTT . . . 91
3.4.2 In directed HoTT . . . 95
3.5 Bridges . . . 105
3.5.1 Discussion . . . 105
3.5.2 Attempts to define bridges . . . 107
3.5.3 An axiomatic treatment of bridges . . . 110
3.6 Homotopies and natural transformations . . . 113
3.7 Groupoids . . . 116
3.8 The higher category structure of some basic types . . . 119
3.8.1 The coproduct . . . 119
3.8.2 The naturals . . . 120
3.8.3 The opposite . . . 122
3.8.4 The core . . . 123
3.8.5 The localization . . . 123
3.8.6 The product . . . 125
3.8.7 Dependent pair types . . . 125
3.8.8 Dependent function types . . . 128
3.8.9 Universe types and univalence . . . 130
3.9 Categorical univalence . . . 133
3.9.1 Category theory in HoTT . . . 133
3.9.2 Categorical univalence . . . 134
3.9.3 Univalence in directed HoTT . . . 136

4 Exploring semantics and consistency 139
4.1 Basic preparations . . . 140
4.2 Interpreting terms . . . 144
4.2.1 Interpreting types . . . 144
4.2.2 Functorial behaviour . . . 147
4.3 Interpreting axioms . . . 148
4.4 Conclusion . . . 149

5 Conclusion 151
5.1 Morphisms and morphisms . . . 151
5.2 Applications . . . 153
5.3 Discussion and further work . . . 153

Bibliography 155

List of Symbols and Abbreviations **157**

Index **161**

Vulgariserende samenvatting

Topologie is voor elastische voorwerpen wat meetkunde is voor starre voorwerpen. In de meetkunde werken we doorgaans in het vlak, de ruimte of een meer-dimensionale veralgemening, en we beschrijven de dingen aan de hand van een assenstelsel, coördinaten en een notie van afstand. In topologie werken we in een topologische ruimte; hier kan je in veel gevallen over nadenken als een verzameling (meer-dimensionale) elastische voorwerpen. Omdat je deze kan uitrekken en vervormen, zijn coördinaten en afstand geen goede manier om topologische ruimten te beschrijven; in de plaats worden open en gesloten verzamelingen gebruikt om de vorm van de ruimte te karakteriseren.

Type theory is een wiskundige formulering van een veiligheidsmechanisme dat gebruikt wordt in statisch getypeerde programmeertalen. In dergelijke talen heeft elke waarde, en elk hokje dat een waarde kan bevatten, een type, en het is slechts toegestaan een waarde in een hokje te steken als de types overeenkomen. Bijvoorbeeld, in een klantendatabase heeft elke klant een geboortejaar en een geboorteplaats. Het geboortejaar heeft als type 'geheel getal', de geboorteplaats 'string' (een opeenvolging van letters). Als de programmeur zich vergist en het geboortejaar toekent aan het hokje voor de geboorteplaats, zal de compiler opmerken dat deze types verschillen, en dit programma niet aanvaarden.

Dependent type theory, een gesofisticeerdere variant van type theory, laat toe om elke wiskundige stelling te vertalen in het type van alle bewijzen van die stelling. Zo kan een programmeur een programma doorspekken met bewijzen over het gedrag van datzelfde programma. Als een bewijs niet correct blijkt, zal de compiler opmerken dat het niet het juiste type heeft en het programma niet aanvaarden. Dit laat de ontwikkeling van veilige software toe, en maakt het ook mogelijk om wiskundige bewijzen te laten verifiëren door een computer.

Categorietheorie is een tak van de wiskunde die de notie van transformaties in hoge algemeenheid bestudeert. Een categorie is gedefinieerd als een collectie objecten, met voor elke twee objecten x en y een verzameling van alle 'morfismen' (transformaties) die x tot y transformeren. Elk object heeft een 'identiek' morfisme naar zichzelf, dat overeenkomt met niets doen. Verder kunnen we een morfisme van x naar y samenstellen met een morfisme van y naar z om een morfisme van x naar z te bekomen. Een hogere categorie bevat niet enkel morfismen tussen objecten, maar ook morfismen tussen morfismen tussen objecten, en morfismen tussen morfismen tussen morfismen tussen objecten, en zo tot in het oneindige. Een ∞-groupoid is een hogere categorie waarin alle morfismen omkeerbaar zijn.

Homotopy type theory (HoTT) is een recente ontwikkeling van dependent type theory die een verband legt tussen topologische ruimten, ∞-groupoids en types. Dit verband laat toe om aan computergeverifieerde topologie en categorietheorie te doen, en leidt anderzijds tot nieuwe inzichten in dependent type theory.

Het doel van deze thesis is om homotopy type theory aan te passen zodat het verband met ∞-groupoids verruimt tot een verband met hogere categorieën in het algemeen. Niet-omkeerbare morfismen, die dus een vaste oriëntatie hebben, krijgen dan een tegenhanger in type theory. Men spreekt van georiënteerde of directed type theory. Dit lijkt de expressiviteit van HoTT aanzienlijk te versterken, wat zowel de software- als wiskundige toepassingen ten goede komt.

English abstract

Type theory is a mathematical formulation of a safety mechanism for programming languages. Dependent type theory (such as Martin-Löf type theory, MLTT) is a more involved variant of type theory that is a sufficiently expressive environment for doing logic. As such, it can serve as a foundation for mathematics [Uni13], and it also brings flexible formal proving to programming languages, which is promising when it comes to writing software for critical applications where bugs could endanger lives.

In MLTT, objects can be judgementally equal, meaning they are absolutely the same and indistinguishable, or propositionally equal, meaning one is 'as good as' the other. Ordinary MLTT basically interprets the latter notion as the former: an object is only equal to itself and only in one way. Homotopy type theory (HoTT) is a recent development of MLTT, which revises the way we think about equality. It starts from the observation that any notion of isomorphism fulfils all the axioms that are required of propositional equality, except the principle of uniqueness of identity proofs [Uni13, §7.2], which states that two objects can only be equal in a single way.

Based on that observation, HoTT replaces uniqueness of identity proofs with the univalence axiom, which asserts that any two equivalent (i.e. isomorphic) types are equal. This axiom turns out to imply a similar principle in more generality, for example [Uni13, §2.14] proves that isomorphic semigroups are equal. That is an extremely useful feature when using type theory as a foundation for mathematics, as we are even using this principle in mathematics founded on set theory, where strictly speaking, it doesn't hold.

The structure of a collection of objects with a sense of isomorphism, is captured by a groupoid: a category in which all morphisms are invertible. This indicates that in HoTT, we should think of types as groupoids. Since the isomorphisms between two objects are also contained in a type, they form a groupoid again. So we are really dealing with groupoids enriched over groupoids enriched over ..., which is called ∞-groupoids. The homotopy hypothesis [Uni13, §2.intro] provides a further interpretation of ∞-groupoids as topological spaces, in which isomorphisms between objects correspond to paths between points. This means that, apart from allowing exciting new ideas and results in type theory, HoTT also allows us to do ∞-groupoid theory and topology in a constructive, computer-verified manner.

Groupoids are a special case of categories and algebraic topology has been generalized to directed algebraic topology [Gra09] with unidirectional paths. It has been investigated how we could modify type theory to create a notion of unidirectional equality (i.e. morphism) [LH11][Lic11][Shu11][War13]. Licata and Harper [LH11][Lic11] restrict the problem to two-dimensional categories. All of the cited approaches are successful at, or have good hopes of proving consistency of the modified theory; all of them have limited success at fully describing functorial behaviour and morphism types with succinct, elegant

and familiar inference rules.

In this thesis, I propose a modification of HoTT that allows us to reason about morphisms, without sacrificing any concepts from the original theory. The core idea is that although we want as many things as possible to be functorial, either co- or contravariant, some functions simply aren't. This leads to the conclusion that we need invariant functions that do not preserve morphisms, although they do preserve propositional equality (isomorphisms) just as functions in HoTT do. A fourth kind of variance, called isovariance, maps morphisms to equality. Combined with dependent types, isovariant functions turn out to be extremely valuable when reasoning about commuting diagrams.

Chapter 1 contains an informal introduction to type theory, homotopy type theory and the work presented in this thesis.

Chapter 2 defines homotopy type theory step by step and immediately generalizes every step to the directed case. Additionally, a few aspects of directed HoTT that are not generalizations of something in symmetric HoTT (which is just a name that we shall use for HoTT in order to emphasize the contrast), such as opposite types and morphism types, are also defined.

Chapter 3 takes the core ideas and results from symmetric HoTT and generalizes them to directed HoTT. Many of them turn out to become richer and sometimes a bit more obscure. There will be two univalence axioms, relating the ∞-groupoid structure on every type induced by propositional equality, the category structure on every type induced by the morphism type, and the category structure on the universe – the type of types – induced by functions. These will be justified from their analogy with the original univalence axiom, and from an analogy with a treatment of categories *within* HoTT given in [Uni13, ch.9].

Chapter 4 contains a stub of a consistency proof, relative to MLTT. It is neither formal nor complete, partially due to time restrictions, partially because we will actually run into problems. I think these problems are not necessarily symptomatic of inconsistency; rather, they indicate that the categorical counterpart of something simple and elegant in type theory, may be extremely complicated.

A peculiar phenomenon that gets some attention in every chapter, are bridges. This is a kind of additional and unexpected structure that appears to be present in every type. It is suspect, because it does not seem to have a counterpart in category theory; zigzags of morphisms

$$a \leftsquigarrow b \rightsquigarrow c \leftsquigarrow d \rightsquigarrow e \leftsquigarrow f$$

seem to be the closest, yet they are quite different. It is unclear to me at this point whether bridges are a genuinely interesting concept, or rather a consequence of a poor design choice in the theory's definition. Surprisingly, they are not the number one problem in the consistency argument in chapter 4. In fact, we can find an ad hoc interpretation of bridges in every type.

Finally, chapter 5 concludes the thesis with a demonstration of the directed version of the idea that isomorphic structures are equal: an informal proof that a group morphism implies a type theoretical morphism; followed by a brief discussion of possible applications and interesting directions for future research.

Nederlandstalige abstract

Typetheorie is een wiskundige formulering van een veiligheidsmechanisme in programmeertalen. Dependent type theory (zoals Martin-Löf type theory, MLTT) is een uitgebreidere variant van typetheorie die voldoende expressief is om mee aan logica te doen. Het kan daardoor dienen als fundering voor de wiskunde [Uni13], en maakt een flexibele vorm van formele bewijzen mogelijk in programmeertalen, wat veelbelovend is met het oog op software voor kritische toepassingen, waar bugs bijvoorbeeld levens in gevaar zouden kunnen brengen.

In MLTT kunnen objecten enerzijds oordeelsmatig gelijk zijn, wat betekent dat ze absoluut dezelfde en ononderscheidbaar zijn; en anderzijds propositioneel gelijk, wat betekent dat ze 'voor elkaar kunnen doorgaan.' Gewone MLTT interpreteert propositionele gelijkheid in feite als oordeelsmatige gelijkheid: een object is alleen gelijk aan zichzelf en dat slechts op één manier. Homotopy type theory (HoTT) is een recente ontwikkeling van MLTT die de manier waarop we over gelijkheid nadenken, herbekijkt. Het uitgangspunt is de observatie dat eenderwelke notie van isomorfisme aan alle axioma's voldoet die we opleggen aan gelijkheid, behalve aan het principe van uniciteit van gelijkheidsbewijzen [Uni13, §7.2], dat stelt dat twee objecten slechts op één enkele manier gelijk kunnen zijn.

Omwille van die observatie vervangt HoTT de uniciteit van gelijkheidsbewijzen door het univalentieaxioma, dat elke twee equivalente (d.w.z. isomorfe) types gelijkstelt. Dit axioma blijkt een gelijkaardig maar algemener principe te impliceren, zo bewijst [Uni13, §2.14] bijvoorbeeld dat isomorfe halfgroepen gelijk zijn. Dat is een erg nuttige eigenschap als we typetheorie als fundering voor de wiskunde willen gebruiken, aangezien we dat principe zelfs toepassen wanneer we wiskunde schragen op verzamelingenleer, hoewel het dan strikt genomen niet geldt.

De structuur van een verzameling objecten met een notie van isomorfisme kan men vatten in het concept van een groupoid: een categorie waarin alle morfismen inverteerbaar zijn. Dit duidt erop dat we types in HoTT moeten zien als groupoids. Aangezien de isomorfismen tussen twee objecten ook bevat zijn in een type, vormen ook zij een groupoid. We hebben dus te maken met groupoids, verrijkt over groupoids verrijkt over ..., wat we ∞-groupoids noemen. De homotopiehyptohese [Uni13, §2.intro] laat een verdere interpretatie van ∞-groupoids als topologische ruimten toe, waarbij isomorfismen tussen objecten overeenkomen met paden tussen punten. Dit betekent dat HoTT niet alleen leidt tot boeiende nieuwe ideeën en resultaten in typetheorie, maar ook toelaat om aan ∞-groupoidtheorie en zelfs topologie te doen op een constructieve, computergeverifieerde manier.

Groupoids zijn een speciaal geval van categorieën en algebraïsche topologie is veralgemeend naar georiënteerde algebraïsche topologie met eenrichtingspaden. Men heeft

onderzocht hoe we typetheorie kunnen aanpassen om een notie van georiënteerde gelijkheid (m.a.w. morfisme) te creëren [LH11][Lic11][Shu11][War13]. Licata en Harper [LH11][Lic11] beperken het probleem tot tweedimensionale categorieën. Alle geciteerde benaderingen van het probleem slagen erin of hebben er een goed oog op dat ze de consistentie van de aangepaste theorie kunnen bewijzen; alle boeken ze minder succes als het erop aankomt functorieel gedrag en morfismetypes te beschrijven met bondige, elegante en vertrouwde inferentieregels.

In deze thesis stel ik een aanpassing van HoTT voor die toelaat om over morfismen te redeneren, zonder enige concepten uit de originele theorie op te offeren. Het centrale idee is dat, hoewel we zoveel mogelijk verbanden functorieel willen houden (co- of contravariant), sommige functies gewoon geen functoren zijn. Dit leidt tot de conclusie dat we invariante functies nodig hebben die geen morfismen bewaren, hoewel ze wel propositionele gelijkheid (isomorfisme) bewaren, net zoals functies in HoTT dat doen. Een vierde soort variantie, genaamd isovariantie, beeldt morfismen af op gelijkheid. In combinatie met dependent types, blijken isovariante functies buitengewoon waardevon in het redeneren over commutatieve diagrammen.

De inleiding in hoofdstuk 1 bevat een informele kennismaking met typetheorie, homotopy type theory en het werk dat in deze thesis wordt uiteengezet.

Hoofdstuk 2 definieert homotopy type theory stap voor stap en veralgemeent onmiddellijk elke stap naar het georiënteerde geval. Verder worden enkele aspecten van georiënteerde HoTT gedefinieerd die geen veralgemening zijn van iets uit symmetrische HoTT (zo zullen we HoTT noemen om het contrast met georiënteerde HoTT te beklemtonen), zoals opposite types en types van morfismen.

Hoofdstuk 3 presenteert de centrale ideeën en resultaten van symmetrische HoTT en veralgemeent deze naar georiënteerde HoTT. Veel ervan blijken rijker te worden en sommige ook iets obscuurder. Er zullen twee univalentieaxioma's nodig zijn, die de ∞-groupoidstructuur op elk type geïnduceerd door propositionele gelijkheid, de categoriestructuur op elk type geïnduceerd door morfismen, en de categoriestructuur op het universum (het type van types) geïnduceerd door functies, aan elkaar relateren.

Hoofdstuk 4 bevat een aanzet tot consistentiebewijs vanuit de consistentie van MLTT. Dit bewijs is noch formeel, noch volledig; deels omwille van tijdgebrek en deels omdat er zich wezenlijke problemen voordoen. Ik denk niet dat die problemen per se symptomen zijn van inconsistentie; ze duiden er wellicht eerder op dat de categorietheoretische tegenhanger van iets wat eenvoudig en elegant is in typetheorie, extreem ingewikkeld kan zijn.

Een merkwaardig fenomeen dat enige aandacht krijgt in elk hoofdstuk, zijn bruggen. Dit is een soort van onverwachte, bijkomdende structuur die aanwezig schijnt te zijn in elk type. Het is verdacht, omdat er geen tegenhanger in categorietheorie te vinden lijkt te zijn; zigzags van morfismen

$$a \leftsquigarrow b \rightsquigarrow c \leftsquigarrow d \rightsquigarrow e \leftsquigarrow f$$

komen het dichtste in de buurt, maar verschillen niettemin wezenlijk. Het is me op dit moment niet duidelijk of bruggen een echt interessant concept zijn, of eerder een gevolg van een suboptimale designkeuze bij het vastleggen van de theorie. Verbazend genoeg zijn bruggen niet het hoofdprobleem in het consistentieargument in hoofdstuk 4. We kunnen er integendeel voor elk type een ad hoc interpretatie voor vinden.

De thesis eindigt in hoofdstuk 5 met een demonstratie van de georiënteerde versie van het idee dat isomorfse structuren gelijk zijn: een informeel bewijs dat een groepsmorfisme een typetheoretisch morfisme impliceert; gevolgd door een korte discussie van mogelijke toepassingen en interessante pistes voor verder onderzoek.

Chapter 1

Introduction

1.1 Dependent type theory

At the most fundamental level, we can only say two things in type theory. We can say that an object a lives in type A, denoted $a : A$, and we can say that two elements a and b of the same type A are judgementally equal: $a \equiv b : A$. These two kinds of assertions are called **judgements**. Both objects a and b, as well as the type A may depend on variables x_i that are assumed to take values in certain types T_i of our choice. These assumptions are an integral part of the judgement: if we want to be explicit, we can write

$$x_1 : T_1, \ldots, x_n : T_n \vdash a : A. \tag{1.1}$$

This means that we cannot, ever, talk about a value without mentioning its type, and we cannot state that a type contains a value without giving it. Moreover, it means that at the most fundamental level, there are no implications, conjunctions, etc. The following claim is therefore nonsense – it is not true or untrue; rather, it cannot be said:

$$\text{If } x \equiv y : A, \text{ then } z : C. \tag{1.2}$$

And yet in the abstract we claimed that MLTT is sufficiently expressive for doing logic and even founding mathematics. How does this work?

The answer is the following: propositions correspond to types. We can encode a proposition as the type of all its proofs, and we can read a type A as the proposition 'A is inhabited'. To see that these are inverses, note that the proposition 'The types of proofs of P is inhabited' is logically equivalent to P, and that the type of proofs that A is inhabited, is simply A.[1]

If propositions can take the form of types, then we need a way to say that a type is "true". There is no specific judgement available for this. Instead, we will just reuse the judgement $x : X$ to say that X is true. So we cannot say "X is true" as such, but we *can* say "x proves X." And when we use the theorem X to prove the theorem Y, then we will have to use the proof x to construct a proof $y : Y$. *This means there is no way to get rid of the proof.* It remains with us forever.

[1] Both claims are disputable; they are intended as informal arguments.

The Curry-Howard correspondence

This section is based on [Uni13, §1.11].

In this section, we will establish the Curry-Howard correspondence between propositions and types. The easiest and least exciting direction is to translate a type into a proposition: whenever we have a type A, we can read it as "A is inhabited." In the rest of this section, we will consider all components of formulas from first order logic and see if we can find a counterpart in MLTT, thus establishing the other direction of the correspondence. As we go, we will also discover the basic types of MLTT, which will be defined more elaborately in chapter 2.

$\top$ The proposition "true" is translated into the type $\mathbf{1}$, which has a single element $\star : \mathbf{1}$. Indeed, it holds, as is proven by $\star$. We could also have translated "true" into the naturals, which hold too (e.g. $\vdash 7 : \mathbb{N}$), but $\mathbf{1}$ has the advantage that it has a unique proof.

$\bot$ The proposition "false" should not have any terms. Hence, we translate it into the empty type $\mathbf{0}$. There is no judgement to say that a proposition does not hold, but the inference rules of MLTT allow us to create an element of any type from an element of $\mathbf{0}$, i.e. to prove any proposition from $\mathbf{0}$. Then at least $\mathbf{0}$ is the falsest type available.

$\wedge$ If A and B are true, then they both have a proof. We put those proofs in a pair and find that the cartesian product $A \times B$ is true. Conversely, if $A \times B$ is true, we use the coordinate projections to find that both A and B are true.

$\Rightarrow$ If A implies B, then surely from a proof of A, we can craft a proof of B. Thus we can build a function $A \to B$. Conversely, if $f : A \to B$ and $a : A$, then $f(a) : B$.

$\Leftrightarrow$ In first order logic, the formula $\varphi \Leftrightarrow \chi$ is equivalent to $(\varphi \Rightarrow \chi) \wedge (\chi \Rightarrow \varphi)$. Similarly, in type theory, we will state "A if and only if B" as the type $(A \to B) \times (B \to A)$. When this type contains a term, we will say that A and B are **logically equivalent**. Because there is more to say about a type than whether or not it has a term, there will be stronger notions of sameness; see section 3.3.

$\vee$ The **coproduct** $A + B$ contains for any element $a : A$ an element $\mathsf{inl}\, a : A + B$, and for any $b : B$ an element $\mathsf{inr}\, b : A + B$. As such, we can prove $A + B$ from either A or B. Conversely, if C holds whenever A or B holds (so that we have proofs $f : A \to C$ and $g : B \to C$), then we can prove $A + B \to C$ by mapping $\mathsf{inl}\, a$ to $f(a)$ and $\mathsf{inr}\, b$ to $g(b)$.

$\neg$ Although we have a notion of truth, there is no obvious notion of falsehood. One might suggest that a type is false if it is not inhabited, but there is no judgement for stating that a type is empty. However, we may state that A implies a contradiction: $A \to \mathbf{0}$. Take heed, though, because there is no general way to create a term of A from a term of $(A \to \mathbf{0}) \to \mathbf{0}$. So in type theory, we generally drop the law of excluded middle $A + (A \to \mathbf{0})$, which is stronger than $((A \to \mathbf{0}) \to \mathbf{0}) \to A$.

$\forall$ How could we formulate the proposition "For every term a of A, it is true that $P(a)$"? The proposition $P(a)$ is a type, so we can see the property P as a function from A

to the universe, which is the type of types: $P : A \to \mathcal{U}$. Then, given $a : A$, we need to find a term $p(a) : P(a)$. So apparently p is a function for which the *type* of the output depends on the *value* of the input. Such functions are called **dependent functions** and their appearance is typical for dependent type theories. The type of dependent functions is denoted using a product sign, $\prod_{a:A} P(a)$, as we can view it as the cartesian product of all types $P(a)$.

$\exists$ How could we formulate "There is an $a : A$ so that $P(a)$"? If we want to prove it, we should give an $a : A$ and a proof $p : P(a)$. So we need a type of pairs, where the *type* of the second component depends on the *value* of the first component. Such pairs are called dependent pairs, and again, they are characteristic for dependent type theories. The type of dependent pairs is denoted using a summation sign, $\sum_{a:A} P(a)$, because we can view it as the coproduct or disjoint union of all types $P(a)$.

$\in$ Here, we arrive at a point where type theoretical logic differs strongly from set theory's first order logic. Whereas $x \in X$ is a proposition as any other, $x : X$ is a judgement and cannot be incorporated in a proposition. It would not be useful anyway, because in type theory, a judgement never contains a term that we do not know the type of. However, in specific cases, there may be an appropriate translation. For example, for a function $f : X \to Y$, the proposition $y \in \mathrm{im} f$ is equivalent to $\exists x \in X : f(x) = y$, which translates into the type $\sum_{x:X} f(x) =_Y y$.

$=$ We need to be able to say that two objects $a, b : A$ are propositionally equal, so apparently there has to be a type $a =_A b$ of proofs that a and b are equal. MLTT usually has **uniqueness of identity proofs**, which asserts that $a =_A b$ can have at most one element. In HoTT, this rule is abandoned in favour of the interpretation of elements of $a =_A b$ as isomorphisms from a to b. If we interpret types as ∞-groupoids, then $a =_A b$ is the Hom-groupoid between a and b. If we think of them as topological spaces, then $a =_A b$ is the path space. These so-called **identity types** are defined in section 2.15.

$\equiv$ Judgemental equality cannot be stated as a proposition, only as a judgement, which is why it is called judgemental equality. Moreover, a type that encodes $a \equiv b$ can be shown to break down the ∞-groupoid structure of types in HoTT.

Proving and programming

In a statically typed programming language such as Java, every value has a type and we have a type checker which gives errors at compile time when we use a value of a certain type in a location where a different type is required. In such a language, we can write expressions, define methods, define classes, define interfaces etc. These things are by no means equivalent.

A functional programming language such as Haskell has first class functions, which means that defining a function becomes a special case of spelling out an expression. However, we still have types and typeclasses, which are fundamentally different from expressions. Calculating a Haskell function has no side effects, which means that if we apply the same function call with the same arguments twice, the returned values must be interchangeable. Then an object is entirely determined by its construction. This is

also the case in MLTT: an object can be seen as a tuple of symbols and these symbols describe how the object was created.

The functional programming language Agda comes even closer to MLTT. It has not only first class dependent functions, but also first class types, so that even defining a type becomes a special case of spelling out an expression. It allows so-called inductive type definitions, which allows programmers to define all the types from the Curry-Howard correspondence and many more, and it has a pattern matching syntax that is analogous to MLTT's rules for creating functions. Typeclasses are not available in Agda and can actually be built using dependent pair types.

To summarize: absolutely the only thing one can do (and would want to do) in Agda is spelling out expressions, have them type checked and give them a name. This sounds a lot like what we are doing in MLTT! In fact, it is as good as the same thing: an Agda-program is an expression and an expression is an object. This is interesting, because a proof is also an object, and if Agda can type check an object, then it can check whether a proof belongs to the type it is supposed to belong to, i.e. whether it proves the theorem it is supposed to prove. So if we found mathematics on MLTT, then software for verifying proofs is already available.

For more about Haskell, see [Lip11]. For more about Agda, see [Nor09] or [BD09].

Proof relevance and constructiveness

A proof of the proposition "For every $a : A$, there exists a $b : B$ so that $P(a, b)$" is a term $p : \prod_{a:A} \sum_{b:B} P(a, b)$. Then given $a : A$, we can actually *compute* a value $b :\equiv \mathsf{prl}(p(a)) : B$ that satisfies $P(a, b)$. This is a remarkable feature of MLTT: every proof is constructive. The price we paid for this, is that we dropped the law of excluded middle. Indeed, if we would postulate an axiom

$$\mathsf{lem} : \prod_{A:\mathcal{U}_i} A + (A \to \mathbf{0}), \tag{1.3}$$

then we can prove many more theorems, but their proofs would contain the ugly term lem which we know nothing about and which obstructs computation. Moreover, it would be inconsistent with the univalence axiom [Uni13, intro].

Remember that there is no way of saying X is true, without saying $x : X$, and that whenever we prove a theorem Y from X, we will be using x in the construction of y. This means that we should think twice before choosing how to prove a theorem. Because suppose we give a proof x for a basic theorem X today, and tomorrow someone uses it to give a proof p of $\prod_{a:A} \sum_{b:B} P(a, b)$ as above. The proof p is constructive and yields a function $f :\equiv \lambda a.\mathsf{prl}(p(a)) : A \to B$. Then the choices we made in constructing x, may influence the output values of the function f.

In type theory, we are doing **proof relevant** mathematics: the fact that a type may have multiple, distinct values, means that a proposition may have multiple substantially different proofs, and it *matters* which proof we give. In terms of programming, we might say that the announcement of a theorem is the declaration of a variable, and the proof is the *definition* of its value.

1.2 Homotopy type theory

Classical MLTT has **uniqueness of identity proofs** [Uni13, §7.2] (or equivalently, axiom K), which states that two objects can be propositionally equal only in one way, i.e. all elements of $a =_A b$ are equal. Using the Curry-Howard correspondence, we can write it as

$$\mathsf{uip} : \prod_{a,b:A} \prod_{p,q:a=b} p =_{a=b} q. \tag{1.4}$$

Here, $\prod_{a,b:A}$ should be seen as an abbreviation of $\prod_{a:A} \prod_{b:A}$. With this rule, we can interpret any type as a set with an equivalence relation: elements of the type are interpreted as elements of the set, judgemental equality is interpreted as being the same, and propositional equality is interpreted as being equivalent.

Homotopy type theory (HoTT) starts from the observation that after removing uip, the inference rules of type theory allow propositional equality to be interpreted as a weak ∞-groupoid structure. A groupoid is a category in which all morphisms are invertible, an ∞-groupoid is roughly a groupoid enriched over the ∞-groupoids. Ours are weak, because they only satisfy the ∞-groupoid laws up to isomorphism. In this interpretation, elements of a type correspond to objects of the ∞-groupoid, judgemental equality corresponds to being the same, identity types correspond to Hom-groupoids, and proofs of equality thus correspond to isomorphisms.

Of course, if we just remove uip, there is a degree of arbitrariness to what this groupoid structure means, as is evident from the fact that it is sound to unremove uip. Voevodsky's univalence axiom determines the meaning of propositional equality by postulating that propositional equality of types is the same as equivalence (a name used in this context for isomorphism). This assertion turns out to be sufficient to conclude a more general these: isomorphism of any kind of structures is the same as propositional equality.

Every topological space has a fundamental ∞-groupoid and every ∞-groupoid has a geometric realization as a topological space. The homotopy hypothesis/theorem (depending on your definition of ∞-groupoids) asserts that these two functors are adjoint and preserve homotopy theory: the subdiscipline of topology that studies only notions that can be formulated in terms of points and paths [Uni13, §2.intro]. This means in particular that it is more or less justified to think of ∞-groupoids, and therefore of types, as topological or at least homotopical spaces. In this interpretation, elements of a type correspond to points in the space, judgemental equality corresponds to being the same, identity types correspond to path spaces, proofs of equality correspond to paths, and a proof that two paths are equal corresponds to a homotopy between them.

As propositional equality is supposed to be a notion of equality, we expect to be able to construct from a function $f : A \to C$, a proof of $(a =_A b) \to (f(a) =_C f(b))$ for all $a, b : A$. Fortunately, the rules of type theory allow us to do this (see section 3.2). In the groupoid interpretation, this means that such functions preserve isomorphisms, which are the only morphisms present in a groupoid. In other words, they are functors. In the homotopical interpretation, this means that functions preserve paths, which can be seen as a notion of continuity.

Of course, in dependent type theory, we also have dependent functions $f : \prod_{a:A} C(a)$. They, too, should preserve equality, but it is a bit less straightforward to state this. Suppose that we have elements $a, b : A$ and a path $p : a =_A b$ (note that we are importing topological terminology to type theory here). Then we would like to prove that $f(a)$ and

$f(b)$ are equal, but they live in different types: $f(a) : C(a)$ and $f(b) : C(b)$. These types are not necessarily judgementally equal, so we cannot state equality just like that. Of course, as $C : A \to \mathcal{U}$ is a non-dependent function from A to the type of types $\mathcal{U}$ (called the universe), we do know that $C(a) =_{\mathcal{U}} C(b)$ and fortunately, one can prove that any two propositionally equal types are also equivalent. Thus, we have a function $p_* : C(a) \to C(b)$ that allows us to transport $f(a)$ to $C(b)$. Then we can state that $p_*(f(a)) =_{C(b)} f(b)$. This will be how dependent functions preserve equality.

Homotopy type theory can serve as a foundation for mathematics, where it provides the principle that isomorphism implies equality, which is absent in set theory, notwithstanding its everyday use by mathematicians. Moreover, it allows us to 'program' topological spaces and in particular do contructive algebraic topology in a computer verified way.

Finally, if computational problems of HoTT can be solved (the univalence axiom gets in the way of computation by asserting that a function is invertible without giving the inverse), then this could allow programmers to switch flexibly between equivalent representations of the same data.

1.3 Directed type theory

There have been some investigations already to modifying type theory so as to broaden the correspondence of types with ∞-groupoids to general higher categories with potentially non-invertible morphisms. The to my knowledge most elaborate and only published work is on two-dimensional directed type theory, by Licata and Harper [LH11][Lic11]. After a brief overview of their work and a mention of other efforts, follows an overview of the approach to directed HoTT taken here and a discussion of what I think are the contributions of this thesis to the subject.

1.3.1 Two-dimensional directed type theory

This part is based on [LH11] and [Lic11, ch.7-8], which also contains research conducted jointly with R. Harper. Notations are adapted to suit better with the rest of this thesis.

Licata and Harper present a theory called two-dimensional directed type theory (2DTT). It is directed, in the sense that for any type A, we can speak of unidirectional morphisms $\varphi : a \rightsquigarrow b$ between terms $a, b : A$ which generalize the notion of invertible paths from HoTT. It is two-dimensional, in the sense that a type A may have several distinct terms, and there may be several distinct morphisms $a \rightsquigarrow b$, but two morphisms $\varphi, \chi : a \rightsquigarrow b$ are either judgementally equal or different. There is no family of objects that express some other particular relation between φ and χ.

Although we can speak of morphisms, they are not contained in a type. Rather, on top of the judgements $a : A$ and $a \equiv b : A$, there are judgements for reasoning about morphisms:

$$\begin{aligned} \varphi : a \rightsquigarrow b : A \qquad & \varphi \text{ is a morphism from } a : A \text{ to } b : A, \\ \varphi \equiv \chi : a \rightsquigarrow b : A \qquad & \varphi \text{ and } \chi \text{ are judgementally equal morphisms from } a : A \text{ to } b : A. \end{aligned}$$

All of these judgements can be stated in a context of variables x_i that are assumed to live in types T_i of our choice. What is notably different from MLTT, is that each of these

variables gets a variance v_i and can only be used in v_i-variant positions. An example of a complete judgement is:

$$x_1 :^{v_1} T_1, \ldots, x_n :^{v_n} T_n \vdash \varphi : a \rightsquigarrow b : A. \tag{1.5}$$

When we have a morphism $\varphi : a \rightsquigarrow a' : A$ and for every $x :^{+} A$ a type $B[x]$, i.e. when $B[x]$ is a type family covariant in x, then we can build from this a function $\varphi_* : B[a] \to B[a']$, where $X \to Y$ is, just as in MLTT, the type of functions from X to Y.[2] However, if for every $x :^{-} A$ we have a type $C[x]$, i.e. when $C[x]$ is a type family contravariant in x, then we get a function $\varphi^* : C[a'] \to C[a]$.

There is a generalization of dependent types. When $B[x]$ is a type family contravariant in $x :^{-} A$, then we can form the type of dependent functions $\prod_{x:A} B[x]$. When for every $x :^{-} A$, $b[x] : B[x]$ is a term depending contravariantly on x, then we can form

$$f :\equiv x \mapsto b[x] : \prod_{x:A} B[x], \tag{1.6}$$

and whenever $\varphi : x \rightsquigarrow y : A$, we will find some $\chi : \varphi^*(b[y]) \rightsquigarrow b[x] : B[x]$.

Consistency of this theory is shown by interpreting every judgement as a statement about the category Cat of categories, where types are interpreted as objects of Cat, i.e. categories, and the elements/morphisms of a type are (roughly) interpreted as objects/morphisms of the corresponding category.

There are a few drawbacks to this generalization of MLTT that will be addressed in by the theory presented in this thesis:

- 2DTT is finite dimensional and therefore it only limitedly accommodates the correspondence between type theory, higher category theory and algebraic topology on which HoTT is based. The authors write encouragingly:

  > "Although it is not necessary for the applications we consider here, it seems likely that 2DTT could be extended to higher dimensions, and that more general interpretations are possible." [LH11]

- The fact that every assumption has to be either co- or contravariant, means that we have to abolish the Martin-Löf identity type. Indeed suppose that $a = \square$ is covariant. Then if we have a morphism $\varphi : a \rightsquigarrow b : A$, it yields a function $\varphi_* : (a = a) \to (a = b)$, so $\varphi_*(\mathsf{refl}\, a)$ proves $a = b$. Then unidirectional morphism implies identity which is precisely what we didn't want to happen. A similar argument shows that $a = \square$ cannot be contravariant.

  In [Lic11, §8.2.5], a way of discretizing types is suggested: for every type A, there would be a type $!A$ which has the same elements but forgets the morphism part of A. Then we could express that elements $a, b : A$ are equal by saying $!a \rightsquigarrow !b : !A$. However, as all functions in 2DTT are either co- or contravariant, there can be no function $!\square : A \to !A$; there is only $!^{-1} : !A \to A$.

  The directed HoTT presented here will remedy this situation by allowing also invariant functions which do not preserve all morphisms. There will be a type A^{core} that is analogous to the discretization found in [Lic11].

[2] In fact, the paper does not explicitly introduce a type of covariant, non-dependent functions. However, at the very least, we may prove $\Gamma, b :^{+} B[a] \vdash \varphi_*(b) : B[a']$, where Γ is the context needed to derive the premises.

- 2DTT does not distinguish between variance of types and variance of their elements. In particular, the judgement $x :^{+} S, y :^{-} T \vdash a : A$ (in which x and y are variable symbols and the rest are expressions) can only be derived when a and A both depend contravariantly on y and covariantly on x. Moreover, because a is covariant in y, the variance of T in x must be opposite to the variance of a and A in x: contravariant. In other words, the variance of T in x, a in y and A in y are all determined by a single boolean.

  To see that this is a problem, suppose we want to derive $x :^{+} S \vdash (y \mapsto c[y]) : \prod_{y:T} C[y]$. As the dependent function type should be contravariant in its domain, this means that T must be contravariant in x so that $\prod_{y:T} C[y]$ depends covariantly on x through T, as required.

  The intended judgement is derived from a judgement of the form $x :^{+} S, y :^{\square} T \vdash c[y] : C[y]$. Since T is contravariant in x, we can only fill out the hole with a minus, for contravariance. This means that all dependent functions have to be contravariant. Indeed, [LH11] only allows the creation of contravariant functions. Covariant dependent functions are suggested in [Lic11, §8.2.4], but they diverge from the familiar notion of covariant functions. In particular, the type itself takes a context and a substitution as arguments, and is covariant in its domain.

  The theory presented in this thesis distinguishes between variance of types and their elements and allows dependent function and pair types of every variance, over type families that possibly have different variance.

- There is no type of morphisms and in particular, there is no induction principle. In section 2.16, I propose a definition of morphism types that is remarkably similar to the definition of identity types.

A clear drawback of the theory described in this thesis, is that consistency is an open problem.

1.3.2 Other related work

There have been other, non-published efforts at developing a directed homotopy type theory. Michael Warren [War13] suggests a rule that allows one to define functions that take a string of three morphisms

$$a \rightsquigarrow b \rightsquigarrow c \rightsquigarrow d \tag{1.7}$$

as argument, by only providing function values for

$$b \overset{\mathsf{id}\, b}{\rightsquigarrow} b \rightsquigarrow c \overset{\mathsf{id}\, c}{\rightsquigarrow} c, \tag{1.8}$$

subject to a few further constraints that make it impossible to prove equality from a morphism. His theory does not keep track of variance judgementally.

Michael Shulman [Shu11] axiomatizes the internal logic of a 2-category. It does not keep track of variance judgementally, as all morphisms in a 2-category behave like covariant functors. As such, there is a functoriality rule that asserts that any function preserves morphisms. Most of the axioms will be provable in directed HoTT as presented here, many are actually proven in chapter 3.

1.3.3 Directed homotopy type theory

The goal of this thesis is to make small and/or elegant modifications to the foundations of homotopy type theory in order to obtain types of morphisms without losing the identity types. This means that there will be two structures on every type: an ∞-groupoid structure induced by the identity type, and a category structure induced by the morphism type. Each of these have a degree of arbitrariness in them. Just like the univalence axiom related the identity types to the category structure induced by functions on the universe, we will have two univalence axioms: one that states that a proof of equality is the same as an isomorphism, and one that states that a morphism between types is the same as a covariant function. This will, again, lead to a more general principle analogous to the idea that isomorphic structures can always be identified: a morphism in the context-dependent sense will be the same as a morphism in the type-theoretic sense, as demonstrated in section 5.1.

Just like 2DTT, we will be keeping track of variance judgementally. As the problems with identity types in 2DTT indicated, we need a third kind of variance on top of co- and contravariance, for functions that are essentially non-functorial: **invariance**.

Calling them non-functorial is a bit too reckless, however: we want any function, even invariant ones, to respect propositional equality, and our univalence axioms state that propositional equality is the same as isomorphism. As such, an invariant function is one that maps isomorphisms to isomorphisms, but potentially discards non-invertible morphisms.

Now that we've broken the taboo of inventing new kinds of variance, why stop? Given objects $a, b : A$ and a function $f : A \to C$, we have five obvious relations between its function values: $f(a) \rightsquigarrow_C f(b)$, $f(a) \leftsquigarrow_C f(b)$, $f(a) =_C f(b)$, $\mathbf{1}$ and $\mathbf{0}$. Covariant functions map a morphism $\varphi : a \rightsquigarrow_A b$ to the first, contravariant functions to the second, and invariant functions to the fourth. Functions that map any morphism to a path/isomorphism, will be called **isovariant**. What about functions that map any non-invertible $\varphi : a \rightsquigarrow_A b$ to $\mathbf{0}$? Such a function is essentially a function of arbitrary variance plus a proof that its domain is a groupoid. Groupoids will be an important concept in directed HoTT and are investigated in sections 2.12 and 3.7.

Just like a contravariant function $A \xrightarrow{-} C$ is the same as a covariant function $A^{\mathsf{op}} \xrightarrow{+} C$, there will be a type A^{core} (the core of A), which is A with all non-invertible morphisms removed, and a type A^{loc} (the localization of A), which is A with all morphisms turned into isomorphisms, so that an invariant function $A \xrightarrow{\times} C$ is the same as a covariant function $A^{\mathsf{core}} \xrightarrow{+} C$, and an isovariant function $A \xrightarrow{=} C$ is the same as a covariant function $A^{\mathsf{loc}} \xrightarrow{+} C$. The operations $\square^{\mathsf{core}}$ and $\square^{\mathsf{loc}}$, seen as 1-functors from the category of categories Cat to the category of groupoids Grpd, are right and left adjoint to the forgetful functor $\mathsf{Grpd} \to \mathsf{Cat}$. [nLa12] This can be seen as an indication that isovariance and invariance are actually the interesting notions to consider.

Variance in higher categories is extremely subtle, however. Consider the functor $\square^{\mathsf{op}}$: it preserves morphisms. Indeed, by the univalence axioms, morphisms between types are covariant functions, and a covariant function $f : A \xrightarrow{+} C$ clearly leads to a covariant function $f' : A^{\mathsf{op}} \xrightarrow{+} C^{\mathsf{op}}$. However, take two covariant functions $f, g : A \xrightarrow{+} C$ and a natural transformation $\nu : f \rightsquigarrow g$. This will give rise to a *reversed* natural transformation $\nu' : g' \rightsquigarrow f'$. So $\square^{\mathsf{op}}$ preserves morphisms *contravariantly*. This indicates that, to fully

express the variance of a function, we need an infinite sequence of tokens in $\{+, -, =, \times\}$. In the remainder of this thesis, we take the following approach, which originally seemed simplifying but in hindsight was not necessarily a good idea:

- A covariant function $A \xrightarrow{+} C$ is a function that is covariant at all levels.
- A contravariant function $A \xrightarrow{-} C$ is a function that reverts morphisms *covariantly*. The corresponding sequence of tokens is $-_0 +_1 +_2 +_3 \ldots$ This is, for example, the variance of $\square \rightsquigarrow b$.
- An isovariant function $A \xrightarrow{=} C$ maps morphisms to proofs of equality. As the identity type can be shown to be a groupoid, higher variance is immaterial.
- An invariant function $A \xrightarrow{\times} C$ discards morphisms, so higher variance, at first sight, is immaterial.
- Functions of any other variance will be weakened to match one of these four. This is always possible, since invariance is the weakest possible variance.

What about dependent functions? In symmetric HoTT (i.e. classical HoTT without directed morphisms), the only relation to be preserved is propositional equality, and whenever $a =_A b$, we know that $C(a) = C(b)$ so that in particular, these types are equivalent and elements can be transported to and fro.

In directed HoTT, a first thing to be noted is that the variance of the function need not match the variance of the type family. For example, it makes sense to consider contravariant functions $f : \prod^{-}_{x:A} C(x)$ for a covariant type family $C : A \xrightarrow{+} \mathcal{U}$. Indeed, if we have a morphism $\varphi : a \rightsquigarrow_A b$, then we get a function $\varphi_* : C(a) \xrightarrow{+} C(b)$ so that $f(a)$ can be transported to $\varphi_*(f(a)) : C(b)$ and contravariance of f should allow us to find a morphism $f(b) \rightsquigarrow_{C(b)} \varphi_*(f(a))$.

Although the variance of C and f do not match in this example, we did critically use the fact that C somehow preserves the structure from A. Can we allow invariant type families?

We first take a look at a special case of invariant functions: if $C(a, b)$ is contravariant in a and covariant in b, then $a \overset{\times}{\mapsto} C(a, a)$ is an invariant function $A \xrightarrow{\times} \mathcal{U}$. This is all but a rare situation: any function that is constructed from co-, contra- and isovariant, but not invariant functions takes this form. Now can we speak of a covariant function $f : \prod^{+}_{x:A} C(x, x)$? In fact, the notion does make sense: from a morphism $\varphi : a \rightsquigarrow_A b$, we get functions

$$C(a, a) \xrightarrow{\varphi_*} C(a, b) \xleftarrow{\varphi^*} C(b, b), \tag{1.9}$$

which allow us to transport $f(a)$ and $f(b)$ to a common type $C(a, b)$. Then the covariance of f should allow us to find a morphism $\varphi_*(f(a)) \rightsquigarrow_{C(a,b)} \varphi^*(f(b))$.

Suppose that $C(A, A)$ is the type of binary operations on A: $C(A, B) :\equiv A \xrightarrow{+} A \xrightarrow{+} B$. Then if we have a morphism, i.e. a covariant function $f : A \xrightarrow{+} B$ and we want to transport operations $* : A \xrightarrow{+} A \xrightarrow{+} A$ and $\odot : B \xrightarrow{+} B \xrightarrow{+} B$ to the common type $A \xrightarrow{+} A \xrightarrow{+} B$ for comparison, then one can show that the transported values become $f(\square * \square)$ and $f(\square) \odot f(\square)$ respectively. This already smells of a general definition of morphisms; promising!

In section 3.4, we show that we can even consider variance of dependent functions into a truly invariant type family, such as $a \overset{\times}{\mapsto} f(a) = g(a)$, which clearly should not preserve morphisms and also does not allow obviously transport to a common type. Therefore, we will allow dependent function and pair types over *any* type family of *any* variance.

The observation that invariant type families C allow us to compare elements of $C(a)$ and $C(b)$ 'along' a morphism $\varphi : a \rightsquigarrow b$, indicates that invariant functions do not entirely discard morphisms and in particular, that A^{core} contains more structure than just the isomorphisms of A. We will call this structure 'bridges' and this thesis contains a lot of pondering and very few formal statements about them.

In particular, this means that we have to consider how functions act on bridges. As the category structure on bridges is not entirely clear at this point, the reasoning about variance will be a bit less thought out when it comes to bridges. The mother of all invariant functions is the function $A \overset{\times}{\to} A^{\mathsf{core}}$, which apparently maps morphisms to bridges. We conclude that invariant functions map morphisms to bridges. Because a composition of invariant functions is again invariant, they should also preserve bridges.

- As invariance has to be the weakest of all variances (so that we can weaken any complicated variance to invariance), we cannot make any assumptions about the variance of the action of invariant functions on bridges and morphisms. We will assume that invariant functions map bridges and morphisms to bridges invariantly.
- It seems reasonable to assume that co- and contravariant functions preserve bridges covariantly.
- Isovariant functions will map bridges to isomorphisms. As the identity type is a groupoid, the variance of the action of isovariant functions on bridges is unimportant.

I believe that the fact that invariant type families map morphisms to relations is a correct and important observation, but the reader will agree that the treatment here in terms of bridges is in many ways unsatisfactory.

1.3.4 Contributions

This thesis contains a systematic and careful analysis of the variance of most basic dependencies in homotopy type theory, and seems to be the first effort of this kind and size. The idea to consider invariance and isovariance as worthy peers of co- and contravariance appears to be novel; invariance appears in other work under the same name, I have not found any account of isovariance. Both concepts pop up naturally when analysing HoTT.

Section 2.16 presents the first inductive definition for a morphism type family, along with an induction principle that matches the definition.

Although a distant analogue may exist in category theory, most likely in the context of Grothendieck fibrations, the observation that invariant type families map morphisms to relations, so that it makes sense to speak of co-, contra- and isovariant dependent functions into invariant type families, is new for a notion of invariance that is as broad as the one we have here.

The absence of a credible consistency proof and the poor understanding of bridges, somewhat reduce the value of the work. However, I hope that the reader will find enough arguments throughout the text to see past these flaws.

Chapter 2

Dependent type theory, symmetric and directed

In the previous chapter, we gave an informal introduction to MLTT, HoTT and directed HoTT. In this chapter and the next one, we will construct a more formal theory for (symmetric) HoTT (heavily based on [Uni13]) and, simultaneously, for directed HoTT. We will do this incrementally, by presenting a morsel of formal HoTT, then adapting this to the directed case, and then moving on to the next morsel. The directed parts are marked with a line in the margin as a reminder of where you are. The main and ever-recurring challenge in these parts is to get the variances right.

In this chapter, we are mostly concerned with basic MLTT which does not especially exploit the groupoid structure of types. It is nevertheless recommendable to keep the groupoid interpretation in mind, as the directed generalization of MLTT will take the higher category structure of types into consideration. We will be mostly defining basic types, postponing the analysis of their behaviour to the next chapter.

The parts about MLTT and HoTT are generally heavily based on [Uni13], whereas the parts about directed HoTT are my own work, sometimes inspired by [LH11].

2.1 Judgements and contexts

This section is based on [Uni13, §1.1 and A.2].

In symmetric HoTT

Presumably the first question one wants to have answered when introduced to type theory is: What is a type? However, the problem is that types, just like sets, are a fundamental concept of their theory and thus have no definition as such. Rather, their properties emerge from the underlying logic. Therefore, it is useful to start this introduction by answering the question: what are the most fundamental assertions, the so called **judgements** we can make in type theory? Before we answer that question, let us ask the same question about set theory with first order logic.

There, the objects we are manipulating in the metatheory are formulas, whereas the sets are simply a metaphor used for making sense of the formulas. For example, the formula

$$\exists z : z \in x \wedge (\forall y : \neg(y \in z)) \tag{2.1}$$

could be read as "The set x has an element z that is empty," and of course the inference rules of first order logic and the axioms of set theory are designed for this interpretation to be sensible, but in the end the formula is just a list of symbols. In first order logic, any formula φ gives rise to a judgement

$$\vdash \varphi \tag{2.2}$$

which should be read "The formula φ is provable", and any judgement is of this form [Uni13, §1.1]. We then have a collection of inference rules that state under what circumstances we may conclude that a judgement holds. For example we have an inference rule

$$\frac{\vdash \varphi \qquad \vdash \varphi \Rightarrow \chi}{\vdash \chi} \tag{2.3}$$

which states that for any formulas φ and χ, we may deduce $\vdash \chi$ if we know that $\vdash \varphi$ and $\vdash \varphi \Rightarrow \chi$.

Now we go back to type theory. There, our judgements take a more complicated form. They may look, for example, like this:

$$x_1 : A_1, \quad x_2 : A_2, \quad \ldots, \quad x_n : A_n \quad \vdash \quad b : B. \tag{2.4}$$

where x_1, …, x_n are variable symbols and A_1, …, A_n, b and B are terms (tuples of symbols). Every term A_i may contain the symbols x_1, …, x_{i-1}; and the terms b and B may contain all variable symbols introduced on the left. The judgement could be pronounced: "When x_1 is an element of type A_1, x_2 is an element of type A_2, …, x_n is an element of type A_n, then b is an element of type B." The part on the left of the $\vdash$ is called the **context**. There are only three kinds of judgements:

$$\Gamma \vdash b : B, \tag{2.5}$$

$$\Gamma \vdash b \equiv c : B, \tag{2.6}$$

$$\Gamma \vdash \mathsf{Ctx}. \tag{2.7}$$

The first one states, just as above, that under the assumptions of the context Γ, b is an element of type B. The second one states that in the context Γ, b and c are **judgementally equal** terms of B, which will have a meaning of being completely identifiable; the same by definition. The third one is more technical and simply states that Γ is a meaningful context.

In addition, we have an impressive collection of inference rules at our disposal to conclude new judgements from old ones. Given the nature of our judgements, it is unsurprising that the inference rules are mainly concerned with modifying the context, building terms and types, and occasionally concluding judgemental equality. In what follows, we will usually not be referring to judgements explicitly, but it is good to know that anything we state in type theory, ultimately has to be formulated as a judgement.

Inference rules are denoted by writing a number of judgements, the premises, above a line, and another judgement, the conclusion, below. On the right, we will sometimes write the rule's name. We begin with two very basic rules. The first one states that the empty context is meaningful:

$$\frac{}{\vdash \mathsf{Ctx}}\,\mathsf{emtpyCtx} \tag{2.8}$$

And the second one states that when, under assumptions Γ, A is a type (i.e. an element of some type of types $\mathcal{U}_k$, see section 2.3), it is meaningful to assume that A has an element:

$$\frac{\Gamma \vdash A : \mathcal{U}_k}{\Gamma, x : A \vdash \mathsf{Ctx}}\mathsf{extendCtx} \tag{2.9}$$

As a convention, any unexplained symbols in an inference rule are universally quantified over. For example, this rule holds for any context Γ.

In directed HoTT

If we want to generalize HoTT to a directed type theory, we will have to account for the variance of assumptions in our judgements. In 2DTT [LH11], there are two possibilities: assumptions are either covariant or contravariant. When a term $b[x] : B$ depends covariantly (or contravariantly) on a variable $x : A$, then from a morphism $\varphi : a \rightsquigarrow a'$ we may conclude that $b[a] \rightsquigarrow b[a']$ (or $b[a'] \rightsquigarrow b[a]$ respectively). However, the identity type $x =_A y$ cannot be co- or contravariant in x or y, because this would allow us to prove identity from a morphism (see section 1.3.1).

Thus, we have to allow for a third kind of variance: we will allow **invariant** assumptions, so that when a term $b[x] : B$ depends invariantly on a variable $x : A$, a morphism $\varphi : a \rightsquigarrow_A a'$ tells us (almost) nothing about $b[a]$ and $b[a']$. Almost, because there will still be a weak connection $b[a] \frown b[a']$ which we will call a **bridge**. This notion is somewhat problematic and is investigated further in section 3.5 and chapter 4. There are identity bridges $a \frown a$ and we can involute a bridge $\beta : a \frown a'$ to obtain $\beta^\dagger : a' \frown a$. A bridge $A \frown B$ between types will be a binary relation between A and B. Additionally, invariant functions do preserve isomorphisms, as we will have a univalence axiom that makes isomorphism coincide with propositional equality, which is always preserved.

In fact, we will also allow a fourth kind of variance: when $b[x] : B$ depends **isovariantly** on $x : A$, then from a morphism $\varphi : a \rightsquigarrow a'$, we may conclude that $b[a] = b[a']$. And there is more: even a bridge $a \frown a'$ will imply that $b[a] = b[a']$.

The following table summarizes how functions of each variance act on different kinds of connections ($a \leftsquigarrow b :\equiv b \rightsquigarrow a$):

		$a \frown b$	$a \rightsquigarrow b$	$a \leftsquigarrow b$	$a = b$
invariant	$\times$	$f(a) \frown f(b)$	$f(a) \frown f(b)$	$f(a) \frown f(b)$	$f(a) = f(b)$
covariant	$+$	$f(a) \frown f(b)$	$f(a) \rightsquigarrow f(b)$	$f(a) \leftsquigarrow f(b)$	$f(a) = f(b)$
contravariant	$-$	$f(a) \frown f(b)$	$f(a) \leftsquigarrow f(b)$	$f(a) \rightsquigarrow f(b)$	$f(a) = f(b)$
isovariant	$=$	$f(a) = f(b)$	$f(a) = f(b)$	$f(a) = f(b)$	$f(a) = f(b)$

Of course, all these actions are functions, e.g. if f is contravariant, we get functions $f^{\rightsquigarrow} : (a \rightsquigarrow b) \to (f(b) \rightsquigarrow f(a))$, so we may ask one level higher: what is the variance of $f^{\rightsquigarrow}$? One can show that the identity type is a groupoid in which all bridges are actually paths, so the question is unimportant for actions that produce paths. Aside of those, we assume that the actions of co- and contravariant functions are covariant (as this is how the type of covariant functions $A \xrightarrow{+} B$ behaves with respect to functions and natural transformations), and that the actions of invariant functions are invariant (as invariance has to be the weakest of all variances). Functions with

other, more complicated variance, will be weakened to one of these four.

Our judgements take completely the same form as in symmetric HoTT, except that every colon gets marked with a variance annotation, like so:

$$x :^{+} A \qquad \text{Object } x \text{ has type } A \text{ and must be used covariantly,} \tag{2.10}$$
$$x :^{-} A \qquad \text{Object } x \text{ has type } A \text{ and must be used contravariantly,} \tag{2.11}$$
$$x :^{\times} A \qquad \text{Object } x \text{ has type } A \text{ and must be used invariantly,} \tag{2.12}$$
$$x :^{=} A \qquad \text{Object } x \text{ has type } A \text{ and must be used isovariantly.} \tag{2.13}$$

Judgemental equalities, too, get a variance annotation, e.g. $a \equiv b :^{+} A$.

It is a commonly known fact in category theory that a contravariant functor $\mathcal{C} \to \mathcal{D}$ is the same as a covariant functor $\mathcal{C}^{\mathsf{op}} \to \mathcal{D}$. Similarly, for every type A we will define a type A^{op} which has the same objects but reversed morphisms. There will be a contravariant invertible function from A to A^{op}, and saying $x :^{-} A$ will be essentially the same as saying $x :^{+} A^{\mathsf{op}}$.

We can do something similar for invariant functions. A less well-known construct is the core of a category, here denoted $\mathcal{C}^{\mathsf{core}}$, which is the category that has the same objects as $\mathcal{C}$, and whose morphisms are the isomorphisms of $\mathcal{C}$ [nLa12]. Of course, these morphisms will also be isomorphisms in $\mathcal{C}^{\mathsf{core}}$, so $\mathcal{C}^{\mathsf{core}}$ is always a groupoid. For any type A, we will define a type A^{core}, which has the same objects and whose morphisms are the isomorphisms of A (which coincide with equality). Additionally, the bridges in A^{core} are the bridges from A (including all morphisms). There will be an invariant invertible function from A to A^{core}, and saying $x :^{\times} A$ will be essentially the same as saying $x :^{+} A^{\mathsf{core}}$. The type A^{core} is the best we can do as a type theoretic counterpart for the objects set of a category.

Another analogue exists for isovariant functions. If $\mathcal{C}$ is a category and W is a set of morphisms in $\mathcal{C}$, then the localization of $\mathcal{C}$ away from W is a category $\mathcal{C}[W^{-1}]$ with a functor $G : \mathcal{C} \to \mathcal{C}[W^{-1}]$ that maps all morphisms in W to isomorphisms. It is characterized by a universal property: if a functor $F : \mathcal{C} \to \mathcal{D}$ maps all morphisms of W to an isomorphism, then there is a unique functor $F' : \mathcal{C}[W^{-1}] \to \mathcal{D}$ so that $F = F' \circ G$ [GZ67]. We will only be using the localization from $\mathcal{C}$ away from *all* morphisms, which we will simply call *the* localization of $\mathcal{C}$. For any type A, we will define a type A^{loc} which has the same objects, but where all morphisms (and indeed all bridges) have been turned into isomorphisms. There will be an isovariant function from A to A^{loc}, but it will not in general be invertible, since A^{loc} has more isomorphisms, and thus more equalities than A, and any function must preserve equality. Therefore, saying $x :^{=} A$ will be a bit different from saying $x :^{+} A^{\mathsf{loc}}$. For more on these matters, see section 2.12.

Variances can be composed. The idea is that $w \circ v$ is the variance of a composition $g \circ f$, where f is v-variant and g is w-variant. This yields the following table:

$$\begin{array}{cccc}
+ \circ + \equiv + & + \circ - \equiv - & + \circ \times \equiv \times & + \circ = \equiv = \\
- \circ + \equiv - & - \circ - \equiv + & - \circ \times \equiv \times & - \circ = \equiv = \\
\times \circ + \equiv \times & \times \circ - \equiv \times & \times \circ \times \equiv \times & \times \circ = \equiv = \\
= \circ + \equiv = & = \circ - \equiv = & = \circ \times \equiv = & = \circ = \equiv =
\end{array} \tag{2.14}$$

Any equation in this table can be verified by going through the previous table twice. Perhaps the most striking equation here is that $= \circ \times \equiv =$. This is because an invariant function doesn't discard morphisms entirely: it weakens them to bridges, which are mapped to equality by an isovariant function. Section 3.5 contains some arguments why this should be the case. Note that composition of variances is commutative.

Remark 2.1.1. The fact that, while introducing the theory, I try to explain the design choices, may make it unclear what is being assumed, what is being defined and what is being proven. The exposition above is about what we *want* to be true. An assumption made here is an assumption that we want something; the proofs prove that we want other things. To get things straight:

- The morphism and identity types have not been defined yet. However, when we define them, we will be able to *prove* that functions of each variance act on them in the desired way.
- Table (2.14) is the *definition* of composition of variances.
- The opposite, core and localization of a type, have not been defined yet. When we define them, we will be able to *prove* their behaviour.

When Γ is a context, then $v \circ \Gamma$ will denote the context where every assumption $x :^w A$ from Γ is replaced with $x :^{v \circ w} A$. By Γ^u we shall mean the context Γ with all variances *replaced* by u.

The empty context is meaningful:

$$\frac{}{\vdash \mathsf{Ctx}}\mathsf{emtpyCtx} \tag{2.15}$$

An isovariant assumption is the weakest assumption, so if some conclusion holds under an isovariant assumption, it holds in any variance. (This is related to the fact that we have covariant functions $A^{\mathsf{core}} \xrightarrow{+} A \xrightarrow{+} A^{\mathsf{loc}}$ and $A^{\mathsf{core}} \xrightarrow{+} A^{\mathsf{op}} \xrightarrow{+} A^{\mathsf{loc}}$.) Similarly, if it holds in any variance, it holds under an invariant assumption. This is expressed by the following rules:

$$\frac{\Gamma, x :^{=} A, \Delta \vdash \mathfrak{X}}{\Gamma, x :^{v} A, \Delta \vdash \mathfrak{X}}\mathsf{dropIso}, \qquad \frac{\Gamma, x :^{v} A, \Delta \vdash \mathfrak{X}}{\Gamma, x :^{\times} A, \Delta \vdash \mathfrak{X}}\mathsf{dropVariance}, \tag{2.16}$$

where $\mathfrak{X}$ quantifies over all possible right hand sides. Note that in this way of expressing inference rules, Δ is a sequence of assumptions, but not a context, as it may depend on the variable x, as well as on the variables from Γ.

When, under assumptions Γ, the object $a : A$ may be used covariantly, then under assumptions $v \circ \Gamma$, it may be used v-variantly. To see this, suppose that $x :^u X$ is one of the assumptions in Γ. Then $a : A$ is u-variant in x. If we now use a in a v-variant function f, then $f(a)$ is v-variant in a and hence $(v \circ u)$-variant in x. In fact, we will stretch this a bit further: if $a : A$ may be used w-variantly under Γ, then it may be

used $(v \circ w)$-variantly under $v \circ \Gamma$:

$$\frac{\Gamma \vdash a :^{w} A}{v \circ \Gamma \vdash a :^{v \circ w} A}\mathsf{composeVar} \tag{2.17}$$

$$\frac{\Gamma \vdash a \equiv b :^{w} A}{v \circ \Gamma \vdash a \equiv b :^{v \circ w} A} \qquad \frac{\Gamma \vdash \mathsf{Ctx}}{v \circ \Gamma \vdash \mathsf{Ctx}} \tag{2.18}$$

The fact that we can compose $\Gamma \vdash \mathsf{Ctx}$ with isovariance and then use the dropping rules, means that we can arbitrarily modify variances in a meaningful context.

The rule composeVar and the dropping rules allow us to derive:

$$\frac{\Gamma \vdash a :^{\times} A}{\Gamma \vdash a :^{v} A} \qquad \frac{\Gamma \vdash a :^{v} A}{\Gamma \vdash a :^{=} A} \tag{2.19}$$

Note that relaxing variance for the conclusion and for the assumptions work in opposite directions: a weaker assumption is more often true.

In most inference rules, we will formulate the conclusion covariantly. If we need to use it with different variance, we should apply composeVar as the *last* step (only followed perhaps by relaxing the variance of the context).

When, under assumptions Γ, A is a type that may be used in any given variance, it is meaningful to assume that A has an element.

$$\frac{\Gamma \vdash A :^{v} \mathcal{U}_k}{\Gamma, x :^{=} A \vdash \mathsf{Ctx}}\mathsf{extendCtx} \tag{2.20}$$

We give the new assumption the variance that makes it the strongest, so that we may relax it. Thanks to composeVar, we can derive the stronger inference rule:

$$\frac{\Gamma \vdash A :^{v} \mathcal{U}_k}{\Gamma^{=}, x :^{=} A \vdash \mathsf{Ctx}} \tag{2.21}$$

by first composing the premise with $=$ and then applying extendCtx.

2.2 Structural inference rules

This section is based on [Uni13, §1.1 and A.2].

In symmetric HoTT

We may state trivial conclusions by echoing assumptions:

$$\frac{\Gamma, x : A, \Delta \vdash \mathsf{Ctx}}{\Gamma, x : A, \Delta \vdash x : A}\mathsf{vble.} \tag{2.22}$$

Due to extendCtx, the following rule is admissible (anything that can be proven using the rule, can be proven without the rule):

$$\frac{\Gamma \vdash A : \mathcal{U}_k \qquad \Gamma, \Delta \vdash \mathfrak{X}}{\Gamma, x : A, \Delta \vdash \mathfrak{X}}\mathsf{wkg.} \tag{2.23}$$

That is, we may weaken our judgement by adding unused assumptions. Similarly, the following is admissible:

$$\frac{\Gamma \vdash a : A \qquad \Gamma, x : A, \Delta[x] \vdash \mathfrak{X}[x]}{\Gamma, \Delta[a] \vdash \mathfrak{X}[a]}\mathsf{subst}. \tag{2.24}$$

That is, when a judgement is true for any $x : A$, it is true for a particular $a : A$. Furthermore, we want judgementally equal terms to be entirely indiscernible. This is expressed by the following rule:

$$\frac{\Gamma \vdash a \equiv a' : A \qquad \Gamma, x : A, \Delta[x] \vdash b[x] : B[x]}{\Gamma, \Delta[a] \vdash b[a] \equiv b[a'] : B[a]}\mathsf{indiscern}. \tag{2.25}$$

Judgemental equality is an equivalence relation:

$$\frac{\Gamma \vdash a : A}{\Gamma \vdash a \equiv a : A} \qquad \frac{\Gamma \vdash a \equiv b : A}{\Gamma \vdash b \equiv a : A} \qquad \frac{\Gamma \vdash a \equiv b : A \qquad \Gamma \vdash b \equiv c : A}{\Gamma \vdash a \equiv c : A} \tag{2.26}$$

Finally, equal types have the same terms:

$$\frac{\Gamma \vdash A \equiv B : \mathcal{U}_k \qquad \Gamma \vdash a : A}{\Gamma \vdash a : B} \qquad \frac{\Gamma \vdash A \equiv B : \mathcal{U}_k \qquad \Gamma \vdash a \equiv a' : A}{\Gamma \vdash a \equiv a' : B} \tag{2.27}$$

In directed HoTT

We may state trivial conclusions by echoing covariant assumptions.

$$\frac{\Gamma, x :^+ A, \Delta \vdash \mathsf{Ctx}}{\Gamma, x :^+ A, \Delta \vdash x :^+ A}\mathsf{vble}. \tag{2.28}$$

It may surprise that we only state this rule for covariant assumptions. However, the other cases are now derivable: Given $\Gamma, x :^v A, \Delta \vdash \mathsf{Ctx}$, we get $\Gamma^=, x :^+ A, \Delta^= \vdash \mathsf{Ctx}$ as modifying variances does not break validity; whence $\Gamma^=, x :^+ A, \Delta^= \vdash x :^+ A$ by the above rule. Using $\mathsf{composeVar}$ (remember that $v \circ = \equiv =$) we get $\Gamma^=, x :^v A, \Delta^= \vdash x :^v A$, which yields the desired conclusion using the dropping rules.

The following rule is admissible:

$$\frac{\Gamma \vdash A :^v \mathcal{U}_k \qquad \Gamma, \Delta \vdash \mathfrak{X}}{\Gamma, x :^w A, \Delta \vdash \mathfrak{X}}\mathsf{wkg}. \tag{2.29}$$

The substitution rule is generalized as follows:

$$\frac{\Gamma \vdash a :^v A \qquad \Gamma, x :^v A, \Delta[x] \vdash \mathfrak{X}[x]}{\Gamma, \Delta[a] \vdash \mathfrak{X}[a]}\mathsf{subst}. \tag{2.30}$$

Indiscernibility becomes:

$$\frac{\Gamma \vdash a \equiv a' :^v A \qquad \Gamma, x :^v A, \Delta[x] \vdash b[x] :^+ B[x]}{\Gamma, \Delta[a] \vdash b[a] \equiv b[a'] :^+ B[a]}\mathsf{indiscern}. \tag{2.31}$$

Again, it is perhaps surprising that we do not state this for $b[x] :^v B[x]$, but only for covariant $b[x]$. The v-variant version should be admissible, however: Given a

derivation that ends with the application of v-variant indiscern, we should be able to find a derivation of the same judgement using only the covariant indiscern by postponing the use of composeVar (with v) until *after* indiscern. From now on, we will not explicitly justify analogous phenomena any more.

Guideline 2.2.1. When we can choose between stating an inference rule that produces a term in arbitrary variance, or stating one that only produces terms that can be used covariantly, we will do the latter without further justification.

Judgemental equality is an equivalence relation:

$$\frac{\Gamma \vdash a :^+ A}{\Gamma \vdash a \equiv a :^+ A} \qquad \frac{\Gamma \vdash a \equiv b :^+ A}{\Gamma \vdash b \equiv a :^+ A} \qquad \frac{\Gamma \vdash a \equiv b :^+ A \qquad \Gamma \vdash b \equiv c :^+ A}{\Gamma \vdash a \equiv c :^+ A} \tag{2.32}$$

Finally, equal types have the same terms:

$$\frac{\Gamma \vdash A \equiv B :^v \mathcal{U}_k \qquad \Gamma \vdash a :^+ A}{\Gamma \vdash a :^+ B} \qquad \frac{\Gamma \vdash A \equiv B :^v \mathcal{U}_k \qquad \Gamma \vdash a \equiv a' :^+ A}{\Gamma \vdash a \equiv a' :^+ B} \tag{2.33}$$

2.3 Universe types

In the rest of this chapter, we will study some important types, including those that occur in the Curry-Howard correspondence, a bit more formally, and try to adapt them to the directed case. There will also be a few types that are specific for the directed case and are not a generalization of an interesting concept in symmetric HoTT. The main challenge in moving from HoTT to directed HoTT, will be to find out the proper variance for every function. In particular, we need to find out how the induction principle's variance can be derived from a type's inductive definition. For more on inductive types, see section 2.7.

In symmetric HoTT

We would like to have a single type $\mathcal{U}$ containing all types. However, from $\mathcal{U} : \mathcal{U}$, one can derive contradictions [Coq92]. To resolve this, we have a tower of universes $\mathcal{U}_0, \mathcal{U}_1, \mathcal{U}_2, \ldots$; one for every natural number, such that every type is in some universe. We don't consider the universes to be a function of the natural numbers, however: the numbers in the subscript are those from the metatheory, not those from HoTT. (Note that there is no type containing all universes, so that a function $n \mapsto \mathcal{U}_n$ cannot be correctly typed.) The following inference rules state how these universes behave [Uni13, §A.2]:

$$\frac{\Gamma \vdash \mathsf{Ctx}}{\Gamma \vdash \mathcal{U}_k : \mathcal{U}_{k+1}}, \qquad \frac{\Gamma \vdash A : \mathcal{U}_k}{\Gamma \vdash A : \mathcal{U}_{k+1}}, \qquad \frac{\Gamma \vdash A : \mathcal{U}_k}{\Gamma, x : A \vdash \mathsf{Ctx}}\mathsf{extendCtx}. \tag{2.34}$$

The first rule states that every universe exists (as an element of the next one), regardless of the context. The second one states that if A is an element of some universe, then it is also an element of the next one. This so called cumulativity of universes is a peculiarity in MLTT, since in general, the type of a term is uniquely determined. The third rule, which we already saw in section 2.1, states that if A is in some universe (which is our

only way of stating that A is a type), then it is meaningful to assume that A contains an element.

Example 2.3.1. The type of natural numbers $\mathbb{N}$ is a basic type that can be built from nothing. Therefore, it lives in the lowest universe: $\mathbb{N} : \mathcal{U}_0$.

Example 2.3.2. A monoid $\mathcal{M}$ consists of a type $M : \mathcal{U}$, a binary operation $* : M \to M \to M$ and a proof p that this operation is associative: $p : \prod_{x,y,z:M}(x * y) * z = x * (y * z)$.[a] However, the underlying type M can live in any universe. In that sense, we can define a type of monoids for every universe:

$$\mathsf{Monoid}_k :\equiv \sum_{M:\mathcal{U}_k} \sum_{*:M\to M\to M} \left(\prod_{x,y,z:M} (x * y) * z = x * (y * z) \right). \tag{2.35}$$

The type Monoid_k uses $\mathcal{U}_k$ as a building block, and this block can be found in $\mathcal{U}_{k+1}$ at the lowest. Therefore, $\mathsf{Monoid}_k : \mathcal{U}_{k+1}$.

[a] The symbol $\prod_{x,y,z:M}$ is a common abbreviation for $\prod_{x:M} \prod_{y:M} \prod_{z:M}$.

In directed HoTT

The rules about universes generalize as follows:

$$\frac{\Gamma \vdash \mathsf{Ctx}}{\Gamma \vdash \mathcal{U}_k :^+ \mathcal{U}_{k+1}}, \qquad \frac{\Gamma \vdash A :^+ \mathcal{U}_k}{\Gamma \vdash A :^+ \mathcal{U}_{k+1}}, \qquad \frac{\Gamma \vdash A :^v \mathcal{U}_k}{\Gamma, x :^= A \vdash \mathsf{Ctx}}\mathsf{extendCtx}. \tag{2.36}$$

The variance in the first rule doesn't matter: if we apply the rule to $\Gamma^= \vdash \mathsf{Ctx}$ and then apply $\mathsf{composeVar}$ with variance v, we get $\Gamma^= \vdash \mathcal{U}_k :^v \mathcal{U}_{k+1}$, after which we can use the variance dropping rules to turn $\Gamma^=$ into Γ. By preserving the variance, the second rule basically states that there is a covariant function from $\mathcal{U}_k$ to $\mathcal{U}_{k+1}$. The third rule comes from section 2.1.

2.4 Non-dependent function types

This section is based on [Uni13, §1.2 and A.2].

In this section, we will introduce a type of non-dependent (ordinary) functions from any type A to any type B. This type is not part of the theory as such, because it is a special case of a type of dependent functions, introduced in the next section. However, we treat non-dependent functions as a warm-up.

In symmetric HoTT

Given types A and B in a common universe, we get a type of functions $A \to B$ in the same universe. This is the **formation rule** for the function types: it tells us how to form the type.

$$\frac{\Gamma \vdash A : \mathcal{U}_k \qquad \Gamma \vdash B : \mathcal{U}_k}{\Gamma \vdash A \to B : \mathcal{U}_k}\mathsf{form}_\to. \tag{2.37}$$

If we can construct a term $b[x]$ for an element of B under the hypothesis that $x : A$, then we can define a function that maps any value $a : A$ to $b[a] : B$. We denote this function as $(x : A) \mapsto b[x]$, or, less formally, $x \mapsto b[x]$.[1] This is the **introduction rule**: it tells us how to introduce elements of the type.

$$\frac{\Gamma, x : A \vdash b[x] : B}{\Gamma \vdash (x : A) \mapsto b[x] : A \to B}\mathsf{intro}_{\to}. \tag{2.38}$$

Whenever we have a function and an element of its domain, we may apply the function to the element. This is the **elimination rule**: it tells us how we can 'eliminate' a function to obtain a value of the codomain.

$$\frac{\Gamma \vdash f : A \to B \qquad \Gamma \vdash a : A}{\Gamma \vdash f(a) : B}\mathsf{elim}_{\to}. \tag{2.39}$$

If we first create a function $f : A \to B$ using $\mathsf{intro}_{\to}$ and then apply it to an argument using $\mathsf{elim}_{\to}$, then we obtain an expression of type B. This expression is by definition the body of the function definition, where we substitute the bound variable with the argument. This is the **computation rule** (or β**-reduction rule**): it tells us how to compute the result of the elimination rule.

$$\frac{\Gamma, x : A \vdash b[x] : B \qquad \Gamma \vdash a : A}{\Gamma \vdash ((x : A) \mapsto b[x])\,(a) \equiv b[a] : B}\mathsf{comp}_{\to}. \tag{2.40}$$

A function contains no more information than its function values. This is the **uniqueness principle** (or η**-expansion rule**).

$$\frac{\Gamma \vdash f : A \to B}{\Gamma \vdash f \equiv ((x : A) \mapsto f(x)) : A \to B}\mathsf{uniq}_{\to}. \tag{2.41}$$

Finally, we want two function definitions with equal body to be equal. The $\mathsf{indiscern}$ rule does not assert this, because it does not take bound variables into account. Therefore, we need an additional rule

$$\frac{\Gamma, x : A \vdash b[x] \equiv b'[x] : B}{\Gamma \vdash (x : A) \mapsto b[x] \equiv (x : A) \mapsto b'[x] : A \to B}\mathsf{indiscern}_{\mapsto}. \tag{2.42}$$

Such a rule has to be given for any symbol that binds a variable, such as for $\prod, \sum, \mathrm{W}$ and most recursion and induction principles.

Definition 2.4.1. Using these rules, we define for any type A, the identity function

$$\mathrm{id}_A :\equiv a \mapsto a : A \to A,$$

and for any types A, B and C the function composition

$$\circ : g \mapsto f \mapsto (x \mapsto g(f(x))) : (B \to C) \to (A \to B) \to (A \to C).$$

[1] A more common notation in computer science is $\lambda(x : A).b[x]$ or $\lambda x.b[x]$. We will use the $\mapsto$ notation which is more common in mathematics, mostly because $\mapsto$ looks better with a variance annotation.

In directed HoTT

Given types A and B in a common universe, and a variance v, we get a type of v-variant functions $A \xrightarrow{v} B$ in the same universe. This type is covariant in B, as we can compose with covariant functions $B \xrightarrow{+} C$ (which will coincide with morphisms) to obtain new v-variant functions $A \xrightarrow{v} C$. As composition $f \circ \square$ with a covariant function f is covariant (a natural transformation $g \rightsquigarrow g'$ yields a natural transformation $f \circ g \rightsquigarrow f \circ g'$), the covariant function type $A \xrightarrow{+} B$ is contravariant in A. However, composition $f \circ \square$ with a v-variant function f, reverts morphisms v-variantly. Thus, $A \xrightarrow{v} B$ is, in general, $(-_0 v_1)$-variant in A. Since $- \equiv -_0 +_1$, we can weaken this to contravariance for $A \xrightarrow{=} B$. However, this does not work for $A \xrightarrow{\times} B$ and $A \xrightarrow{-} B$, so we have to see these types as invariant in A.

$$\frac{\Gamma \vdash A :^u \mathcal{U}_k \qquad \Gamma \vdash B :^+ \mathcal{U}_k \\ u :\equiv - \text{ if } v \in \{+, =\}, \text{ and } u :\equiv \times \text{ if } v \in \{-, \times\}}{\Gamma \vdash A \xrightarrow{v} B :^+ \mathcal{U}_k}\mathsf{form}_{\to}. \tag{2.43}$$

If we can construct a term $b[x]$ for an element of B under the hypothesis that $x :^v A$ (so we have used x only v-variantly), then we can define a v-variant function that maps any value $a : A$ to $b[a] : B$. We denote this function as $(x : A) \xmapsto{v} b[x]$, or, less formally, $x \xmapsto{v} b[x]$.

$$\frac{\Gamma \vdash A :^= \mathcal{U}_k \qquad \Gamma \vdash B :^= \mathcal{U}_k \\ \Gamma, x :^v A \vdash b[x] :^+ B}{\Gamma \vdash (x : A) \xmapsto{v} b[x] : A \xrightarrow{v} B}\mathsf{intro}_{\to}. \tag{2.44}$$

Note that we added two premises that weren't there in the symmetric case: we required that A and B be types. In the symmetric case, these premises are clearly redundant, as we can always prove them when the third premise is provable. In fact here, they are redundant as well, but we include them to show that function definition is isovariant in the domain and the codomain. This can be understood in two ways.

Firstly, note that when we use $\mathsf{extendCtx}$ to push a type behind the colon, there are no restrictions on the variance. That is, we may even use types that can only be used isovariantly. This suggests the following guideline:

Guideline 2.4.2 (Sheer existence)**.** Asserting the sheer existence of a type or type family, falls under isovariant use.

There is another argument that may be more convincing but which should have an analogue in any case where Guideline 2.4.2 can be applied. We will show that two function definitions that are identical in every respect except the domain, are equal when there is a morphism between the domains. If that is true, we can say that function definition is isovariant in the domain. So suppose that we have a covariant function (i.e. a morphism) $f : A \xrightarrow{+} A'$. We construct two functions $g :\equiv (x : A) \xmapsto{v} b[x] : A \xrightarrow{v} B$ and $g' :\equiv (x : A') \xmapsto{v} b'[x] : A' \xrightarrow{v} B$ with the 'same' body, and we have to demonstrate that g equals g'. However, there is a problem

with this notion of sameness, as $b[x]$ takes $x : A$ and $b'[x]$ takes $x : A'$. To be able to compare them, we have to use f to 'transport' $b'[x]$ to become a term that takes an argument from A. (While transporting elements between types is a well-defined and precise notion, transporting open terms (terms containing a variable) is not. However, the argument we are giving is not and needn't be formal: we define our inference rules as we please, I'm just trying to explain why this one is sensible). So let us assume that, for all $x : A$, $b[x] = b'[f(x)]$. This is in fact what we meant earlier when we said that g and g' had the 'same' body. We now have to prove that g equals g'. Again, they have different types. So we transport g' to $A \xrightarrow{+} B$ by composing with f, and have to prove that $g = g' \circ f$. But $g(a) = b[x] = b'[f(x)]$ and $(g' \circ f)(x) = b'[f(x)]$, which completes the argument.

Whenever we have a v-variant function and an element of its domain that can be used v-variantly, we may apply the function to the element:

$$\frac{\Gamma \vdash A :^{=} \mathcal{U}_k \qquad \Gamma \vdash B :^{=} \mathcal{U}_k \qquad \Gamma \vdash f :^{+} A \xrightarrow{v} B \qquad \Gamma \vdash a :^{v} A}{\Gamma \vdash f(a) :^{+} B}\mathsf{elim}_{\to}. \tag{2.45}$$

Again, we include redundant premises as a reminder of isovariance in A and B.

The computation rule and the uniqueness principle bring no surprises:

$$\frac{\Gamma \vdash A :^{=} \mathcal{U}_k \qquad \Gamma \vdash B :^{=} \mathcal{U}_k \qquad \Gamma, x :^{v} A \vdash b[x] :^{+} B \qquad \Gamma \vdash a :^{v} A}{\Gamma \vdash \Big((x : A) \xmapsto{v} b[x]\Big)\,(a) \equiv b[a] :^{+} B}\mathsf{comp}_{\to}. \tag{2.46}$$

$$\frac{\Gamma \vdash A :^{=} \mathcal{U}_k \qquad \Gamma \vdash B :^{=} \mathcal{U}_k \qquad \Gamma \vdash f :^{+} A \xrightarrow{v} B}{\Gamma \vdash f \equiv \Big((x : A) \xmapsto{v} f(x)\Big) :^{+} A \xrightarrow{v} B}\mathsf{uniq}_{\to}. \tag{2.47}$$

Neither does the indicernibility rule:

$$\frac{\Gamma, x :^{v} A \vdash b[x] \equiv b'[x] :^{+} B}{\Gamma \vdash (x : A) \xmapsto{v} b[x] \equiv (x : A) \xmapsto{v} b'[x] :^{+} A \xrightarrow{v} B}\mathsf{indiscern}_{\mapsto}. \tag{2.48}$$

Definition 2.4.3. Using these rules, we define for any type $A :^{=} \mathcal{U}_k$ the identity function

$$\mathrm{id}_A :\equiv a \xmapsto{+} a : A \xrightarrow{+} A, \tag{2.49}$$

and for any types $A, B, C :^{=} \mathcal{U}_k$ the function composition

$$\circ : g \xmapsto{+} f \xmapsto{w} \Big(x \xmapsto{w \circ v} g(f(x))\Big) : (B \xrightarrow{w} C) \xrightarrow{+} (A \xrightarrow{v} B) \xrightarrow{w} (A \xrightarrow{w \circ v} C). \tag{2.50}$$

Note the peculiar variances: as function application is covariant in the applied function, $g(f(x))$ is covariant in g and w-variant in f, because f occurs in the argument of the w-variant function g.

2.5 Dependent function types

This section is based on [Uni13, §1.4 and A.2].

In symmetric HoTT

Dependent functions are a generalization of ordinary functions. When $B[x]$ is a type depending on a variable $x : A$, we have a type $\prod_{x:A} B[x]$ of functions f which map any term $a : A$ to a term $f(a) : B[a]$. So the type of the output depends on the value of the input. We use a product sign because the type of dependent functions may also be regarded as the cartesian product of all the types $B[x]$.

Example 2.5.1. When $n : \mathbb{N}$ is a natural number and $A : \mathcal{U}_i$ is a type, we have a type $\mathsf{Vec}_n\, A$ of A-tuples of length n. Then we could construct a function $f : \prod_{n:\mathbb{N}} \mathsf{Vec}_n\, \mathbb{N}$ which maps $n : \mathbb{N}$ to the vector $f(n) = (0, 1, 2, \ldots, n-1) : \mathsf{Vec}_n\, \mathbb{N}$, e.g. $f(3) = (0, 1, 2) : \mathsf{Vec}_3\, \mathbb{N}$.

Example 2.5.2. In type theory, the usual way to represent a function in more than one argument, is in **curried**[a] form. For example, if f takes $a : A$ and $b : B$ and yields $f(a, b) : C$, then we see f as a term of the type $A \to (B \to C)$, rather than $(A \times B) \to C$. Then $f(a, b)$ really means $f(a)(b)$. We take the arrow to be right associative, so we write simply $f : A \to B \to C$. Similarly, we take the product sign to apply to everything on its right, so the dependent generalization of this is $f : \prod_{a:A} \prod_{b:B[a]} C[a, b]$.

As an example of a dependent function in multiple arguments, we might define a function that takes a type $A : \mathcal{U}_i$, then a natural number $n : \mathbb{N}$ and a vector $v : \mathsf{Vec}_{n+1}\, A$ and yields the vector's zeroth component:

$$\begin{aligned} g &: \prod_{A:\mathcal{U}_i} \prod_{n:\mathbb{N}} \mathsf{Vec}_{n+1}\, A \to A, \\ g(B) &: \prod_{n:\mathbb{N}} \mathsf{Vec}_{n+1}\, B \to B, \\ g(B, 3) &: \mathsf{Vec}_4\, B \to B, \\ g(B, 3, (b_0, b_1, b_2, b_3)) &\equiv b_0 : B. \end{aligned} \tag{2.51}$$

[a] Currying, after Haskell Curry, is the process of writing a function $A \times B \to C$ as a function $A \to (B \to C)$. The reverse process is called **uncurrying**.

Whenever we have a type A and for every $x : A$ a type $B[x]$, both in the same universe, then there is a type of dependent functions $\prod_{x:A} B[x]$, also in that universe:

$$\frac{\Gamma \vdash A : \mathcal{U}_k \qquad \Gamma, x : A \vdash B[x] : \mathcal{U}_k}{\Gamma \vdash \prod_{x:A} B[x] : \mathcal{U}_k}\mathsf{form}_\Pi. \tag{2.52}$$

Whenever we have, for all $x : A$, an element $b[x] : B[x]$, we can create a dependent function:

$$\frac{\Gamma, x : A \vdash b[x] : B[x]}{\Gamma \vdash (x : A) \mapsto b[x] : \prod_{x:A} B[x]}\mathsf{intro}_\Pi. \tag{2.53}$$

Whenever we have a dependent function and a term from its domain, we may apply the function to the term:

$$\frac{\Gamma \vdash f : \prod_{x:A} B[x] \qquad \Gamma \vdash a : A}{\Gamma \vdash f(a) : B[a]}\mathsf{elim}_{\prod}. \tag{2.54}$$

If we first create a dependent function using $\mathsf{intro}_{\prod}$ and then apply it to an argument a using $\mathsf{elim}_{\prod}$, then we obtain a value of type $B[a]$. This is by definition equal to the body of the function definition, where we substitute the bound variable with the argument.

$$\frac{\Gamma, x : A \vdash b[x] : B[x] \qquad \Gamma \vdash a : A}{\Gamma \vdash ((x : A) \mapsto b[x])\,(a) \equiv b[a] : B[a]}\mathsf{comp}_{\prod}. \tag{2.55}$$

A dependent function contains no more information than its function values:

$$\frac{\Gamma \vdash f : \prod_{x:A} B[x]}{\Gamma \vdash f \equiv ((x : A) \mapsto f(x)) : \prod_{x:A} B[x]}\mathsf{uniq}_{\prod}. \tag{2.56}$$

Finally, we need to generalize the indiscernibility rule:

$$\frac{\Gamma, x : A \vdash b[x] \equiv b'[x] : B[x]}{\Gamma \vdash (x : A) \mapsto b[x] \equiv (x : A) \mapsto .b'[x] : \prod_{x:A} B[x]}\mathsf{indiscern}_{\mapsto}. \tag{2.57}$$

When B does not depend on the variable x, we abbreviate $\prod_{x:A} B$ as $A \to B$.

In directed HoTT

When comparing the formation rule for $\to$ in the directed case and the formation rule for Π in the symmetric case, one finds that we need to answer one question: what variance can the type family $B[x]$ have in x? For invariant dependent functions it doesn't matter: an equality within the domain always yields a transport. As explained in section 1.3.3, it is sensible for any variance v to allow the formation of $\prod^{v}_{z:A} B[z, z]$ where the type family $B[x, y]$ is contravariant in x and covariant in y. In fact, we will allow families $B[x]$ that are invariant in x. In this case, a morphism $x \rightsquigarrow y$ will create a bridge $B[x] \frown B[y]$ which will still allow us to compare elements of $B[x]$ and $B[y]$. Our formation rule thus becomes:

$$\frac{\begin{array}{l}\Gamma \vdash A :^{-} \mathcal{U}_k \qquad \Gamma, x :^{\times} A \vdash B[x] :^{+} \mathcal{U}_k \\ u :\equiv - \text{ if } v \in \{+, =\}, \text{ and } u :\equiv \times \text{ if } v \in \{-, \times\}\end{array}}{\Gamma \vdash \prod^{v}_{x:A} B[x] :^{+} \mathcal{U}_k}\mathsf{form}_{\prod}. \tag{2.58}$$

In the other rules again we include the redundant premise that A and $B[x]$ exist and can be used isovariantly. The variance of $B[x] :^{=} \mathcal{U}_k$ in $x : A$ does not matter, as we can compose that premise with isovariance and use the variance dropping rules to arbitrarily modify the variance annotations in its context.

$$\frac{\begin{array}{l}\Gamma \vdash A :^{=} \mathcal{U}_k \qquad \Gamma, x :^{\times} A \vdash B[x] :^{=} \mathcal{U}_k \\ \Gamma, x :^{v} A \vdash b[x] :^{+} B[x]\end{array}}{\Gamma \vdash (x : A) \overset{v}{\mapsto} b[x] :^{+} \prod^{v}_{x:A} B[x]}\mathsf{intro}_{\prod}. \tag{2.59}$$

$$\frac{\Gamma \vdash A :^{=} \mathcal{U}_k \qquad \Gamma, x :^{\times} A \vdash B[x] :^{=} \mathcal{U}_k \qquad \Gamma \vdash f :^{+} \prod^{v}_{x:A} B[x] \qquad \Gamma \vdash a :^{v} A}{\Gamma \vdash f(a) :^{+} B[a]}\mathsf{elim}_{\prod}. \tag{2.60}$$

$$\frac{\Gamma \vdash A :^{=} \mathcal{U}_k \qquad \Gamma, x :^{\times} A \vdash B[x] :^{=} \mathcal{U}_k \qquad \Gamma, x :^{v} A \vdash b[x] :^{+} B[x] \qquad \Gamma \vdash a :^{v} A}{\Gamma \vdash \left((x : A) \overset{v}{\mapsto} b[x]\right)(a) \equiv b[a] :^{+} B[a]}\mathsf{comp}_{\prod}. \tag{2.61}$$

$$\frac{\Gamma \vdash A :^{=} \mathcal{U}_k \qquad \Gamma, x :^{\times} A \vdash B[x] :^{=} \mathcal{U}_k \qquad \Gamma \vdash f :^{+} \prod^{v}_{x:A} B[x]}{\Gamma \vdash f \equiv \left((x : A) \overset{v}{\mapsto} f(x)\right) :^{+} \prod^{v}_{x:A} B[x]}\mathsf{uniq}_{\prod}. \tag{2.62}$$

$$\frac{\Gamma, x :^{v} A \vdash b[x] \equiv b'[x] :^{+} B[x]}{\Gamma \vdash (x : A) \overset{v}{\mapsto} b[x] \equiv (x : A) \overset{v}{\mapsto} b'[x] :^{+} \prod^{v}_{x:A} B[x]}\mathsf{indiscern}_{\mapsto}. \tag{2.63}$$

When B does not depend on the variable x, we abbreviate $\prod^{v}_{x:A} B$ as $A \overset{v}{\to} B$.

2.6 Intermezzo: equivalences and function extensionality

In symmetric HoTT

In the following sections, we will often need a sense of invertible functions. Giving a proper definition for invertibility is not trivial in HoTT, see section 3.3. For now, we will restrict ourselves to a logical definition:

Definition 2.6.1. A function $f : A \to B$ is an equivalence if and only if there exists a function $g : B \to A$ so that $g \circ f = \mathrm{id}_A$ and $f \circ g = \mathrm{id}_B$. Formally:

$$\mathsf{isEquiv}(f) \leftrightarrows \sum_{g:B\to A} (g \circ f = \mathrm{id}_A) \times (f \circ g = \mathrm{id}_B), \tag{2.64}$$

where $X \leftrightarrows Y$ (X if and only if Y) abbreviates $(X \to Y) \times (Y \to X)$.

Note that this is not a complete definition: it does not define the internal structure of $\mathsf{isEquiv}(f)$. Indeed, the fact that there exists a function in every direction only asserts that $\mathsf{isEquiv}(f)$ is inhabited if and only if the right hand side is inhabited. So they are logically equivalent, but not equivalent as types. For a real definition of equivalence, see section 3.3.

If there is a function $f : A \to B$ that is an equivalence, we say that A and B are equivalent: $A \simeq B$. Formally [Uni13, §2.4]:

$$A \simeq B :\equiv \sum_{f:A\to B} \mathsf{isEquiv}(f). \tag{2.65}$$

In order to prove that two types are equivalent, we will need to prove that some functions are equal to the identity function. However, there happens to be a problem: even when we know that two functions $f, g : A \to B$ are pointwise equal, we cannot prove

(just yet [Uni13, §2.9]) that they are equal. To solve this problem, we assume the function extensionality axiom. Putting it too simply, it has type

$$\mathsf{funExt} : \prod_{f,g:A\to B} \left(\prod_{a:A} f(a) =_B g(a) \right) \to (f = g). \tag{2.66}$$

We treat function extensionality in more detail in section 3.8.8.

In directed HoTT

In the directed case, the logical definition looks like this:

Definition 2.6.2. A function $f : A \overset{+}{\to} B$ is an equivalence if and only if there exists a function $g : B \overset{+}{\to} A$ so that $g \circ f = \mathrm{id}_A$ and $f \circ g = \mathrm{id}_B$. Formally:

$$\mathsf{isEquiv}(f) \overset{+}{\leftrightarrows} \sum_{g:B\overset{+}{\to}A}^{+} (g \circ f = \mathrm{id}_A) \times (f \circ g = \mathrm{id}_B), \tag{2.67}$$

where $X \overset{+}{\leftrightarrows} Y$ abbreviates $(X \overset{+}{\to} Y) \times (Y \overset{+}{\to} X)$. The type $\mathsf{isEquiv}(f)$ is covariant in f and invariant in A and B.

For a proper definition, see section 3.3.

The type of equivalences is defined as above, but is now denoted $A \overset{+}{\simeq} B$, because we will sometimes informally mention equivalences of different variance.

Function extensionality in the directed case takes the following form:

$$\mathsf{funExt} : \prod_{f,g:A\to B}^{=} \left(\prod_{a:A}^{=} f(a) =_B g(a) \right) \to (f = g). \tag{2.68}$$

Again, see section 3.8.8 for details.

2.7 Inductive types and the coproduct

This section draws ideas from [Uni13, ch.5, §1.7 and §A.2].

In symmetric HoTT

Typically, type theory comes with a wide range of available types and inference rules describing their behaviour. These inference rules are not entirely arbitrary, however: they reflect a structure that most of these basic types have in common. The **inductive definition** scheme captures this structure. An inductive definition has no formal meaning, at least not in the type theory presented here, but can be seen as an abbreviation of the inference rules describing the type. Alternatively, most (if not all) inductively defined types (or **inductive types**) can be constructed up to equivalence (a kind of isomorphism between types, see section 3.3) by using a smaller range of basic types, including most notably the W-type (see section 2.17).

In order to define a new inductive type C, we have to state how one can build terms of C. We do this by providing zero or more constructors, each one taking zero or more arguments and yielding values in C. The type C is then seen as being "freely generated" by these constructors [Uni13, §5.1]. For example:

Inductive type 2.7.1. If A and B are types, we define their **coproduct** (or disjoint union) $A + B$ as the inductive type with the following constructors:

- $\mathsf{inl} : A \to A + B$,
- $\mathsf{inr} : B \to A + B$.

Note that, to be precise, we should subscript the constructors $\mathsf{inl}_{A,B}$ and $\mathsf{inr}_{A,B}$. When the subscript is clear from the context, we will avoid this, however. The formation rule states that, whenever A and B are types in the same universe, we get a type $A + B$ in that universe:

$$\frac{\Gamma \vdash A : \mathcal{U}_k \qquad \Gamma \vdash B : \mathcal{U}_k}{\Gamma \vdash A + B : \mathcal{U}_k}\mathsf{form}_+ \tag{2.69}$$

Note that we can see $+$ as a function:

$$A \mapsto B \mapsto A + B : \mathcal{U}_k \to \mathcal{U}_k \to \mathcal{U}_k. \tag{2.70}$$

The inductive definition gives us two ways to construct an element of $A+B$: given $a : A$, we have $\mathsf{inl}(a) : A+B$, and given $b : B$, we have $\mathsf{inr}(b) : A+B$. This yields two introduction rules:

$$\frac{\Gamma \vdash A : \mathcal{U}_k \qquad \Gamma \vdash B : \mathcal{U}_k \qquad \Gamma \vdash a : A}{\Gamma \vdash \mathsf{inl}\, a : A + B}\mathsf{introl}_+, \tag{2.71}$$

$$\frac{\Gamma \vdash A : \mathcal{U}_k \qquad \Gamma \vdash B : \mathcal{U}_k \qquad \Gamma \vdash b : B}{\Gamma \vdash \mathsf{inr}\, b : A + B}\mathsf{intror}_+. \tag{2.72}$$

Note that although the inductive definition presents inl and inr as functions, the inference rules show that they are actually primitive operators. However, we can easily define the corresponding functions $a \mapsto \mathsf{inl}\, a$ and $b \mapsto \mathsf{inr}\, b$, which go by the same name.

Now how do we build a function $A + B \to C$? Just like $\mathsf{intro}_\prod$ allows us to build a function $A \to C$ if we have a function value for every element of a, the **recursion principle** rec_{A+B} allows us to build a function $A + B \to C$ if we have a function value for every element of the form $\mathsf{inl}\, a$ and every element of the form $\mathsf{inr}\, b$:

$$\frac{\begin{array}{l}\Gamma \vdash C : \mathcal{U}_k \\ \Gamma, a : A \vdash r[a] : C \\ \Gamma, b : B \vdash s[b] : C \\ \Gamma \vdash e_0 : A + B\end{array}}{\Gamma \vdash \mathsf{rec}_{A+B}(C, a.r[a], b.s[b], e_0) : C}\mathsf{elimNonDep}_+ \tag{2.73}$$

The term 'recursion principle' will become clear when we treat lists, see section 2.8. Note that the notation rec_{A+B} is slightly misleading: it depends on A and B, but not on $A+B$. We should see $\mathsf{rec}_{\square+\square}$ as a single character. The recursion principle is an operator, but we can capture its functionality in a function:

$$(C : \mathcal{U}_k) \mapsto (f : A \to C) \mapsto (g : B \to C) \mapsto (e : A + B) \mapsto \mathsf{rec}_{A+B}(C, a.f(a), b.g(b), e)$$

$$: \prod_{C:\mathcal{U}_k} (A \to C) \to (B \to C) \to (A + B \to C) \tag{2.74}$$

which we give the same name. Since the recursion principle operator binds variables, we need special indiscernibility rules, which we omit.

When we apply the recursion principle to a value of the form $\mathsf{inl}\, a$ or $\mathsf{inr}\, b$, the result is defined as one would expect. We only state the left computation rule:

$$\frac{\begin{array}{l}\Gamma \vdash C : \mathcal{U}_k \\ \Gamma, a : A \vdash r[a] : C \\ \Gamma, b : B \vdash s[b] : C \\ \Gamma \vdash a_0 : A\end{array}}{\Gamma \vdash \mathsf{rec}_{A+B}(C, a.r[a], b.s[b], \mathsf{inl}\, a_0) \equiv r[a_0] : C}\mathsf{complNonDep}_+ \tag{2.75}$$

In fact, we do not need the above elimination and computation rules, they are special cases of their dependent counterparts. To build a dependent function $\prod_{e:A+B} C[e]$, we need to give, for every $a : A$, a function value $r[a] : C[\mathsf{inl}\, a]$ for $\mathsf{inl}\, a$, and for every $b : B$ a function value $s[b] : C[\mathsf{inr}\, b]$ for $\mathsf{inr}\, b$:

$$\frac{\begin{array}{l}\Gamma, e : A + B \vdash C[e] : \mathcal{U}_k \\ \Gamma, a : A \vdash r[a] : C[\mathsf{inl}\, a] \\ \Gamma, b : B \vdash s[b] : C[\mathsf{inr}\, b] \\ \Gamma \vdash e_0 : A + B\end{array}}{\Gamma \vdash \mathsf{ind}_{A+B}(e.C[e], a.r[a], b.s[b], e_0) : C[e_0]}\mathsf{elim}_+ \tag{2.76}$$

The operator ind_{A+B} is called the **induction principle** for the coproduct. Again, we refer to the section on the natural numbers for an explanation of this terminology. Again, we omit the indiscernibility rule. A similar function definition captures the induction principle operator in a function with the same name:

$$\mathsf{ind}_{A+B} : \prod_{C:A+B\to\mathcal{U}_k} \left(\prod_{a:A} C(\mathsf{inl}\, a)\right) \to \left(\prod_{b:B} C(\mathsf{inl}\, b)\right) \to \left(\prod_{e:A+B} C(e)\right). \tag{2.77}$$

Note that the recursion principle can be defined from the induction principle:

$$\mathsf{rec}_{A+B}(C) :\equiv \mathsf{ind}_{A+B}(e \overset{\times}{\mapsto} C). \tag{2.78}$$

Here is the left computation rule:

$$\frac{\begin{array}{l}\Gamma, e : A + B \vdash C[e] : \mathcal{U}_k \\ \Gamma, a : A \vdash r[a] : C[\mathsf{inl}\, a] \\ \Gamma, b : B \vdash s[b] : C[\mathsf{inr}\, b] \\ \Gamma \vdash a_0 : A\end{array}}{\Gamma \vdash \mathsf{ind}_{A+B}(e.C[e], a.r[a], b.s[b], \mathsf{inl}\, a_0) \equiv r[a_0] : C[\mathsf{inl}\, a_0]}\mathsf{compl}_+. \tag{2.79}$$

A uniqueness principle can be added, but is of lesser importance as, for inductive types, it can be proven internally to hold propositionally. We follow [Uni13] and impose no uniqueness principle.

From this experience, we extract guidelines for constructing inference rules from an inductive definition:

Guideline 2.7.2 (Inference rules for an inductive type T)**.**

- The premises of the formation rule, state what objects we need (in this case, we needed types A and B). Its conclusion states that the inductively defined type, lives in the lowest universe that contains all types mentioned in the premises. This is enforced by using the same universe $\mathcal{U}_k$ throughout the inference rule.
- Every constructor $\chi_i : A_1 \to \ldots \to A_n \to T$ yields an introduction rule that takes elements of every A_i as premises and yields an element of T. If necessary, we also add premises to make sure that our type T exists. E.g. above, we explicitly required that A and B were types, because the third premise mentions only either A or B. (Indeed, in this case there was some redundancy in the premises). From this inference rule we can construct a function that has the type $A_1 \to \ldots \to A_n \to T$ mentioned in the inductive definition.
- The elimination rule has one premise $\Gamma, e : T \vdash C[t] : \mathcal{U}_k$ and one premise $\Gamma \vdash e_0 : T$. On top of that, for every constructor $\chi_i : A_1 \to \ldots \to A_n \to T$, there is a premise $\Gamma, a_1 : A_1, \ldots, a_n : A_n \vdash r_i[a_1, \ldots, a_n] : C[\chi_i(a_1, \ldots, a_n)]$. The conclusion is a term of type $C[e_0]$ that wraps up all of this information, called the induction principle. We can turn the induction principle into a function which has an argument for every premise.
- For every constructor χ_i, there is a computation rule which has the same premises as the elimination rule, except the one about e_0. Instead, it also takes the premises of χ_i (except perhaps those that would be redundant). As a conclusion, it states that the induction principle evaluated in $\chi_i(a_1, \ldots, a_n)$ equals $r_i[a_1, \ldots, a_n]$.

In directed HoTT

We now seek to generalize the coproduct, and indeed the concept of an inductive definition, to the directed case. It is now best to think of A and B as (higher) categories.

Inductive type 2.7.3. If A and B are types, we define their **coproduct** $A + B$, which is covariant in A and B, as the inductive type with the following constructors:

- $\mathsf{inl} : A \xrightarrow{+} A + B$,
- $\mathsf{inr} : B \xrightarrow{+} A + B$.

The variance of the constructors is a choice (although choosing a different one may have unexpected consequences for the variance of the inductive type, see the opposite type in section 2.12). We might have taken inl contravariant, which would have given us the type $A^{\mathsf{op}} + B$.

What about the variance of the inductive type itself, though? Remember that we

intend to arrange things so that morphisms between types coincide with covariant functions. Thus, we have to consider what functions $\alpha : A \to A'$ and $\beta : B \xrightarrow{+} B'$ mean to the coproducts $A + B$, $A' + B$ and $A + B'$. It is easy to see that we can expect functions $A + B \to A' + B$ and $A + B \to A + B'$. The reason is that the elements of $A+B$ are constructed from elements of the arguments of the constructor, which are covariant in A and B (since they *are* A and B in this case). Thus, we can easily push morphisms through the coproduct former $+$. This leads to the following restriction:

Guideline 2.7.4. If an inductive type is v-variant in an object x, then the types of all arguments of all constructors must be v-variant in x. If one of the constructors as a function is not covariant in some argument, there are further restrictions; see section 2.12.

The formation rule states that, whenever A and B are types that may be used covariantly, we get a type $A + B$:

$$\frac{\Gamma \vdash A :^+ \mathcal{U}_k \qquad \Gamma \vdash B :^+ \mathcal{U}_k}{\Gamma \vdash A + B :^+ \mathcal{U}_k}\mathsf{form}_+ \tag{2.80}$$

Note that we can see $+$ as a function:

$$A \xmapsto{+} B \xmapsto{+} A + B : \mathcal{U}_k \xrightarrow{+} \mathcal{U}_k \xrightarrow{+} \mathcal{U}_k. \tag{2.81}$$

The introduction rules become:

$$\frac{\Gamma \vdash A :^= \mathcal{U}_k \qquad \Gamma \vdash B :^= \mathcal{U}_k \qquad \Gamma \vdash a :^+ A}{\Gamma \vdash \mathsf{inl}\, a :^+ A + B}\mathsf{introl}_+, \tag{2.82}$$

$$\frac{\Gamma \vdash A :^= \mathcal{U}_k \qquad \Gamma \vdash B :^= \mathcal{U}_k \qquad \Gamma \vdash b :^+ B}{\Gamma \vdash \mathsf{inr}\, b :^+ A + B}\mathsf{intror}_+. \tag{2.83}$$

The covariance of the conclusion is in line with Guideline 2.2.1 on page 20. Covariance in a and b simply reflects the variance of the constructors. The isovariance in the types follows from Guideline 2.4.2, but as another illustration of this guideline's validity, we give a full argument. We will justify the isovariance in A for both introduction rules; the isovariance in B follows by symmetry of the problem.

Let $f : A \xrightarrow{+} A'$ be a covariant function (i.e. a morphism). Then f lifts to a function g which takes $\mathsf{inl}\, a$ to $\mathsf{inl}\, f(a)$ and $\mathsf{inr}\, b$ to itself. We have:

$$\mathsf{inl}_{A+B} : A \xrightarrow{+} A + B, \qquad \mathsf{inl}_{A'+B} : A' \xrightarrow{+} A' + B. \tag{2.84}$$

In order to compare them, we need to transport them to a common type:

$$g \circ \mathsf{inl}_{A+B} : A \xrightarrow{+} A' + B, \qquad \mathsf{inl}_{A'+B} \circ f : A \xrightarrow{+} A' + B. \tag{2.85}$$

Now observe that $g(\mathsf{inl}_{A+B} a) = \mathsf{inl}_{A'+B} f(a)$ as we said only moments ago. This shows that isovariance is the right choice.

The elimination rule becomes:

$$\frac{\begin{array}{l}\Gamma \vdash A :^{=} \mathcal{U}_k \qquad \Gamma \vdash B :^{=} \mathcal{U}_k \\ \Gamma, e :^{\times} A + B \vdash C[e] :^{=} \mathcal{U}_k \\ \Gamma, a :^{v} A \vdash r[a] :^{+} C[\mathsf{inl}\, a] \\ \Gamma, b :^{v} B \vdash s[b] :^{+} C[\mathsf{inr}\, b] \\ \Gamma \vdash e_0 :^{v} A + B\end{array}}{\Gamma \vdash \mathsf{ind}^{v}_{A+B}(e.C[e], a.r[a], b.s[b], e_0) :^{+} C[e_0]}\mathsf{elim}_+ \tag{2.86}$$

Note first of all that we allow an induction principle for every variance, which constructs v-variant dependent functions from $A + B$ to the type family $C[e]$.

The premises about A and B weren't there in the symmetric case and are in fact redundant, but are included here to stress that the induction principle is isovariant in both types. The type family $C[e]$, too, is there just for asserting its existence, so it, too, is being used isovariantly.

Let f be the function that we are trying to construct. Then $f \circ \mathsf{inl} = a \overset{?}{\mapsto} r[a]$. Since we want f to be a v-variant function and inl is covariant, we should have a v-variant function on the right. This justifies that $r[a]$ is v-variant in a (and that $s[b]$ is v-variant in b). The function f is obtained by pasting together $r[a]$ and $s[b]$ and pasting together is covariant (indeed it preserves natural transformations), which explains the variances for $r[a]$ and $s[b]$. Finally, f is v-variant, which is reflected by the variance of e_0.

Note that we can still write the induction principle as a function:

$$\mathsf{ind}^{v}_{A+B} : \overset{=}{\prod_{C:A+B \overset{\times}{\to} \mathcal{U}_k}} \left(\prod_{a:A}^{v} C(\mathsf{inl}\, a)\right) \overset{+}{\to} \left(\prod_{b:B}^{v} C(\mathsf{inl}\, b)\right) \overset{+}{\to} \left(\prod_{e:A+B}^{v} C(e)\right). \tag{2.87}$$

Here is the left computation rule:

$$\frac{\begin{array}{l}\Gamma \vdash A :^{=} \mathcal{U}_k \qquad \Gamma \vdash B :^{=} \mathcal{U}_k \\ \Gamma, e :^{\times} A + B \vdash C[e] :^{=} \mathcal{U}_k \\ \Gamma, a :^{v} A \vdash r[a] :^{+} C[\mathsf{inl}\, a] \\ \Gamma, b :^{v} B \vdash s[b] :^{+} C[\mathsf{inr}\, b] \\ \Gamma \vdash a_0 :^{v} A\end{array}}{\Gamma \vdash \mathsf{ind}_{A+B}(e.C[e], a.r[a], b.s[b], \mathsf{inl}\, a_0) \equiv r[a_0] :^{+} C[\mathsf{inl}\, a_0]}\mathsf{compl}_+. \tag{2.88}$$

All premises but the last are taken from the elimination rule. The constructor inl is covariant and $\mathsf{inl}\, a_0$ takes the role of e_0, so a_0 is being used $(v \circ +) \equiv v$-variantly.

Again, we extract guidelines:

Guideline 2.7.5 (Inference rules for an inductive type T)**.** The only modification of the inference rules compared to the symmetric case (see Guideline 2.7.2 page 31), is the addition of variance annotations. The conclusion of all inference rules can be used covariantly.

- The variance of the formation rule can be read from the definition and is re-

stricted by Guideline 2.7.4.

- The constructors and the introduction rules are isovariant in the building blocks of the inductive type. The variance in the arguments of the constructor, can be read from the definition.
- The v-variant induction principle is isovariant in the building blocks of the inductive type and also in the destination type family. The destination type family may be invariant in its argument. The induction principle is covariant in the functions that it pastes together. If a constructor is u-variant in a given argument, then the pasted function is $(v \circ u)$-variant in that argument. The resulting function is v-variant.
- The computation rules for the v-variant induction principle, preserve the variances of the premises from the elimination rule, and compose the variances of the premises from the introduction rules with v.

2.8 Lists

The type $\mathsf{List}\, A$ consists of finite lists of elements of A. It does not appear in the Curry-Howard correspondence and neither is it one of basic the types that we need to mimic any inductive type (see section 2.17). The main reason for including it here is that it has a recursive constructor, just like the naturals, but unlike the naturals this type is well suited for investigating the implications in the directed case.

In symmetric HoTT

Inductive type 2.8.1. Given a type A, we define the type $\mathsf{List}\, A$ of **lists of** A with constructors:

- $\ddagger : \mathsf{List}\, A$,
- $:: : A \to \mathsf{List}\, A \to \mathsf{List}\, A$.

The object $\ddagger$ is called the **empty list** and $::$ (called **cons** and used as an infix operator) adds an element to the head of a list.

What is new here, is that the constructor $::$ takes an argument of the type that we are defining. This is not a circular definition. In general, we allow the following:

Guideline 2.8.2. A constructor of an inductive type T may take *arguments* of type $X_1 \to \ldots \to X_n \to T$, where the types X_i should not contain any references to T. (One says that T can occur only **strictly positively** in the arguments' types.) E.g. the following constructor is allowed:

- $\chi : A \to (B \to T) \to T \to (C \to T) \to T$,

but the following is not:

- $\psi : (T \to A) \to T$.

For an explanation of this guideline, see [Uni13, §5.6].

In order to construct a function $f : \mathsf{List}\, A \to C$, we need a function value for the empty list, and a function value for $a :: \alpha$ *assuming that* $f(\alpha)$ *is already defined*. For example, if f is the function that takes a list such as $a :: b :: c :: \ddagger$ and returns $a :: a :: b :: b :: c :: c :: \ddagger$, then we will say:

- $f(\ddagger) :\equiv \ddagger$,
- if $f(\alpha) \equiv \gamma$, then $f(a :: \alpha) :\equiv a :: a :: \gamma$.

Thus, the (non-dependent) elimination rule becomes:

$$\frac{\begin{array}{l}\Gamma \vdash C : \mathcal{U}_k \\ \Gamma \vdash r : C \\ \Gamma, a : A, \alpha : \mathsf{List}\, A, \gamma : C \vdash s[a, \alpha, \gamma] : C \\ \Gamma \vdash e_0 : \mathsf{List}\, A\end{array}}{\Gamma \vdash \mathsf{rec}_{\mathsf{List}}(r, a.\alpha.\gamma.s[a, \alpha, \gamma], e_0) : C}\,\mathsf{elim}_{\mathsf{List}} \tag{2.89}$$

or as a function:

$$\mathsf{rec}_{\mathsf{List}\, A} : \prod_{C:\mathcal{U}_k} C \to (A \to \mathsf{List}\, A \to C \to C) \to (\mathsf{List}\, A \to C)\,. \tag{2.90}$$

The function described above would be defined as:

$$\mathsf{rec}_{\mathsf{List}\, A}(\mathsf{List}\, A, \ddagger, a \mapsto \alpha \mapsto \gamma \mapsto a :: a :: \gamma). \tag{2.91}$$

The computation rules are summarized as follows:

$$\mathsf{rec}_{\mathsf{List}\, A}(C, r, s)(\ddagger) \equiv r, \tag{2.92}$$

$$\mathsf{rec}_{\mathsf{List}\, A}(C, r, s)(a :: \alpha) \equiv s\left(a, \alpha, \mathsf{rec}_{\mathsf{List}\, A}(C, r, s)(\alpha)\right). \tag{2.93}$$

This also explains the term 'recursion principle': it allows us to define functions recursively.

The induction principle is:

$$\mathsf{ind}_{\mathsf{List}\, A} : \prod_{C:\mathsf{List}\, A \to \mathcal{U}_k} C(\ddagger) \to \left(\prod_{a:A} \prod_{\alpha:\mathsf{List}\, A} C(\alpha) \to C(a :: \alpha)\right) \to \left(\prod_{e:\mathsf{List}\, A} C(e)\right), \tag{2.94}$$

with identical computation rules.

The following guideline expresses how the recursion and induction principles treat recursive constructors:

Guideline 2.8.3. Suppose we have an inductive type T and want to build a function $g : T \to C$. By Guideline 2.7.2, the recursion principle takes a function f_i for every constructor χ_i. In case the constructor χ_i has a recursive argument of type $X_1 \to \ldots \to X_n \to T$, then the function f_i takes not only an argument $\alpha : X_1 \to \ldots \to X_n \to T$, but also an argument $\gamma : X_1 \to \ldots \to X_n \to B$, used for passing the 'already defined' values $g(\alpha(x_1, \ldots, x_n))$ to the recursion step f_i.

The guideline for the induction principle is analogous, but cumbersome to write

in full generality. The example for List is most likely more instructive.

In directed HoTT

Inductive type 2.8.4. Given a type A, we define the type $\mathsf{List}\,A$, which is covariant in A, with constructors:

- $\ddagger : \mathsf{List}\,A$,
- $:: : A \xrightarrow{+} \mathsf{List}\,A \xrightarrow{+} \mathsf{List}\,A$.

Since $\mathsf{List}\,A$ is defined to be covariant in A, the argument types of the constructors (none for $\ddagger$ and A and $\mathsf{List}\,A$ for $::$) must be covariant in A as per Guideline 2.7.4. We are free to choose the constructors' variance, and choosing them covariant yields the most reasonable behaviour. From Guideline 2.7.5, we conclude that both $\ddagger$ and $::$ are isovariant in A. The v-variant induction principle, which is also isovariant in A, is:

$$\mathsf{ind}^{v}_{\mathsf{List}\,A} : \prod_{C:\mathsf{List}\,A \xrightarrow{\times} \mathcal{U}_k}^{=} C(\ddagger) \xrightarrow{+} \left(\prod_{a:A}^{+} \prod_{\alpha:\mathsf{List}\,A}^{+} C(\alpha) \xrightarrow{+} C(a :: \alpha) \right) \xrightarrow{+} \left(\prod_{e:\mathsf{List}\,A}^{v} C(e) \right). \tag{2.95}$$

All variances here, are prescribed by Guideline 2.7.5, except the one that says that the recursion step $C(\alpha) \xrightarrow{+} C(a :: \alpha)$ is covariant. This can be seen from the following commutative diagram:

$$\begin{array}{ccc} \alpha : \mathsf{List}\,A & \xmapsto{a::} & a :: \alpha : \mathsf{List}\,A \\ \Big\downarrow{\scriptstyle f} & \circlearrowright & \Big\downarrow{\scriptstyle f} \\ f(\alpha) : C(\alpha) & \xmapsto[\text{step}]{\text{recursion}} & f(a :: \alpha) : C(a :: \alpha) \end{array} \tag{2.96}$$

where f is the function being constructed by the induction principle. Since this function is v-variant, we get $v \circ + = w \circ v$ where w is the variance of the recursion step. We deduce the following guideline:

Guideline 2.8.5. If a constructor is u-variant in a recursive argument, then the corresponding function passed to the induction principle is also u-variant in the corresponding 'already defined' value.

2.9 The naturals

See also [Uni13, §1.9].

Just like the type of lists, the type of naturals $\mathbb{N}$ is neither part of the Curry-Howard correspondence, nor necessary for mimicking general inductive types. We include it here

because it is a particularly useful type.

In symmetric HoTT

Inductive type 2.9.1. We define the type of **natural numbers** $\mathbb{N}$ with the following constructors:

- $\mathsf{zero} : \mathbb{N}$,
- $\mathsf{succ}\, n : \mathbb{N} \to \mathbb{N}$.

Note that the constructors of $\mathbb{N}$ match those of $\mathsf{List}\,\mathbf{1}$, where $\mathbf{1}$ is the type that contains only the element $\star$. The only difference is that $::$ takes an extra argument of type $\mathbf{1}$, but that doesn't convey any information so we needn't bother. One can in fact prove that the types are equivalent (there is an invertible correspondence, see section 3.3): $\mathbb{N} \simeq \mathsf{List}\,\mathbf{1}$. We will often abbreviate elements of $\mathbb{N}$ using Arabic numbers, e.g. instead of

$$\mathsf{succ}\,(\mathsf{succ}\,(\mathsf{succ}\,(\mathsf{succ}\,(\mathsf{succ}\,(\mathsf{succ}\,(0)))))), \tag{2.97}$$

we would write 6.

The induction principle is the same as for $\mathsf{List}\,\mathbf{1}$, with arguments of type $\mathbf{1}$ removed:

$$\mathsf{ind}_{\mathbb{N}} : \prod_{C:\mathbb{N}\to\mathcal{U}_k} C(\mathsf{zero}) \to \left(\prod_{n:\mathbb{N}} C(n) \to C(\mathsf{succ}\, n)\right) \to \left(\prod_{n:\mathbb{N}} C(n)\right). \tag{2.98}$$

It computes:

$$\begin{aligned} \mathsf{ind}_{\mathbb{N}}(C, c_0, c_{\mathrm{step}})(\mathsf{zero}) &\equiv c_0 : C(\mathsf{zero}), \\ \mathsf{ind}_{\mathbb{N}}(C, c_0, c_{\mathrm{step}})(\mathsf{succ}\, n) &\equiv c_{\mathrm{step}}(n, \mathsf{ind}_{\mathbb{N}}(C, c_0, c_{\mathrm{step}})(n)) : C(\mathsf{succ}\, n). \end{aligned} \tag{2.99}$$

Suppose we want to prove a proposition $C(n)$ for all natural numbers $n : \mathbb{N}$. Since propositions are types, C is just a type family $C : \mathbb{N} \to \mathcal{U}_k$. A universal quantifier translates to a Π-type, so we want to prove the statement $\prod_{n:\mathbb{N}} C(n)$. This can be done using the induction principle: we just need to prove $C(\mathsf{zero})$ and, for all $n : \mathbb{N}$, derive $C(\mathsf{succ}(n))$ from $C(n)$. This explains the name 'induction principle': it allows us to prove theorems by induction.

In directed HoTT

Inductive type 2.9.2. We define the type of **natural numbers** $\mathbb{N}$ as the inductive type with the following constructors:

- $\mathsf{zero} : \mathbb{N}$,
- $\mathsf{succ}\, n : \mathbb{N} \xrightarrow{+} \mathbb{N}$.

The only thing that may need an explanation here, is the choice to make succ covariant. Firstly, this is in line with the correspondence between $\mathbb{N}$ and $\mathsf{List}\, A$. Secondly, there are no non-trivial morphisms, so indeed they are preserved. Of course this argument works in defence of any variance, and indeed we will show that there is a

succ-function of any variance.

The induction principle is:

$$\mathsf{ind}^v_{\mathbb{N}} : \prod_{C:\mathbb{N}\overset{\times}{\to}\mathcal{U}_k}^{=} C(0) \overset{+}{\to} \left(\prod_{n:\mathbb{N}}^{v} C(n) \overset{+}{\to} C(\mathsf{succ}\, n)\right) \overset{+}{\to} \left(\prod_{n:\mathbb{N}}^{v} C(n)\right), \tag{2.100}$$

as prescribed by Guideline 2.7.5 and Guideline 2.8.5. The computation rules bring nothing new:

$$\begin{aligned} \mathsf{ind}^v_{\mathbb{N}}(C, c_0, c_{\mathrm{step}})(0) &\equiv c_0 : C(0), \\ \mathsf{ind}^v_{\mathbb{N}}(C, c_0, c_{\mathrm{step}})(\mathsf{succ}\, n) &\equiv c_{\mathrm{step}}(n, \mathsf{ind}^v_{\mathbb{N}}(C, c_0, c_{\mathrm{step}})(n)) : C(\mathsf{succ}\, n). \end{aligned} \tag{2.101}$$

Now the reader may question the value of getting variance right for functions of natural numbers, as we don't expect any non-trivial morphisms in $\mathbb{N}$. The reader has a point: we can construct an isovariant identity function $\mathrm{id}_{\mathbb{N}} : \mathbb{N} \overset{=}{\to} \mathbb{N}$. Then from a function $f : \prod^v_{n:\mathbb{N}} C(n)$ of arbitrary variance, we can build a function $f \circ \mathrm{id}_{\mathbb{N}} : \prod^{=}_n C(n)$, since $v \circ = \equiv =$. As isovariance is the strongest variance, any isovariant function is also co-, contra- and invariant. Thus, we will conclude that variance is entirely irrelevant for functions from the natural numbers. We will use 4 as a variance annotation as a reminder of this irrelevance.

We construct the isovariant identity function by isovariant induction. To $\mathrm{id}_{\mathbb{N}}(0)$, we assign the value 0. Then for all $n :^{=} \mathbb{N}$, we get a value $m :^{+} \mathbb{N}$ for $\mathrm{id}_{\mathbb{N}}(n)$ and need to give a value for $\mathrm{id}_{\mathbb{N}}(\mathsf{succ}\, n)$. The simplest thing is to set $\mathrm{id}_{\mathbb{N}}(\mathsf{succ}\, n) :\equiv \mathsf{succ}\, n$. However, n must be used isovariantly and succ is only covariant. The solution is to use recursion: we set $\mathrm{id}_{\mathbb{N}}(\mathsf{succ}\, n) :\equiv \mathsf{succ}\, m$, as m can be used covariantly. Formally:

$$\mathrm{id}_{\mathbb{N}} :\equiv \mathsf{rec}^{=}_{\mathbb{N}}\left(\mathbb{N}, 0, n \overset{=}{\mapsto} m \overset{+}{\mapsto} \mathsf{succ}\, m\right) : \mathbb{N} \overset{=}{\to} \mathbb{N}. \tag{2.102}$$

2.10 The family of finite sets

See also [Uni13, §1.3, 1.5 and 1.7].

In symmetric HoTT

Inductive type 2.10.1. We define the **empty type 0** as the inductive type with no constructors.

In order to construct a function $\mathbf{0} \to C$, we have to provide a function for every constructor, i.e. we needn't provide anything:

$$\mathsf{rec}_{\mathbf{0}} : \prod_{C:\mathcal{U}_i} (\mathbf{0} \to C). \tag{2.103}$$

There are zero computation rules: one for every constructor. The type $\mathbf{0}$ will represent the proposition "false" and then this recursion principle is the principle "ex falso quodlibet":

anything from the absurd. The induction principle is quite similar:

$$\mathsf{ind}_{\mathbf{0}} : \prod_{C:\mathbf{0}\to\mathcal{U}_i} \left(\prod_{x:\mathbf{0}} C(x) \right). \tag{2.104}$$

Again, there are no computation rules.

Inductive type 2.10.2. We define the **unit type 1** as the inductive type with the following constructor:

- $\star : \mathbf{1}$.

In order to construct a function $f : \mathbf{1} \to C$, we have to give $f(\star)$:

$$\mathsf{rec}_{\mathbf{1}} : \prod_{C:\mathcal{U}_i} C \to (\mathbf{1} \to C). \tag{2.105}$$

The computation rule is $\mathsf{rec}_{\mathbf{1}}(C, c)(\star) \equiv c : C$. The induction principle is similar:

$$\mathsf{ind}_{\mathbf{1}} : \prod_{C:\mathbf{1}\to\mathcal{U}_i} C(\star) \to \left(\prod_{x:\mathbf{1}} C(x) \right), \tag{2.106}$$

with computation rule $\mathsf{ind}_{\mathbf{1}}(C, c)(\star) \equiv c : C(\star)$.

Definition 2.10.3. We define the function $\mathsf{Fin} : \mathbb{N} \to \mathcal{U}_0$, called the **family of finite sets**, recursively: $\mathsf{Fin}(0) :\equiv \mathbf{0}$ and $\mathsf{Fin}(\mathsf{succ}\, n) :\equiv \mathsf{Fin}(n) + \mathbf{1}$. Formally:

$$\mathsf{Fin} :\equiv \mathsf{rec}_{\mathbb{N}}(\mathcal{U}_0, \mathbf{0}, n \mapsto X \mapsto (X + \mathbf{1})). \tag{2.107}$$

The element $\mathsf{inr}(\star) : \mathsf{Fin}(n)$ will be denoted $(n-1)_n$ and $\mathsf{inl}(m_{n-1}) : \mathsf{Fin}(n)$ will be denoted m_n. Thus, the terms of $\mathsf{Fin}(n)$ are $0_n, \ldots, (n-1)_n$. This is just a matter of notation and does not require type-theoretical justification.

Example 2.10.4. One application of the family of finite sets, is for indexing fixed-length vectors. We could define a general coordinate projection

$$\pi : \prod_{X:\mathcal{U}_i} \prod_{n:\mathbb{N}} \mathsf{Vec}_n\, X \to \mathsf{Fin}(n) \to X, \tag{2.108}$$

so that for example $\pi(A, 3, (a_0, a_1, a_2), 2_3) \equiv a_2 : A$.

In directed HoTT

Inductive type 2.10.5. We define the **empty type 0** as the inductive type with no constructors.

The recursion principle is:

$$\mathsf{rec}_{\mathbf{0}}^{v} : \prod_{C:\mathcal{U}_i}^{=} (\mathbf{0} \xrightarrow{v} C). \tag{2.109}$$

The induction principle is:

$$\mathsf{ind}_{\mathbf{0}}^{v} : \prod_{C:\mathbf{0}\xrightarrow{\times}\mathcal{U}_i}^{=} \left(\prod_{x:\mathbf{0}}^{v} C(x) \right). \tag{2.110}$$

There are no computation rules. We can define an isovariant identity function:

$$\mathrm{id}_{\mathbf{0}} :\equiv \mathsf{rec}_{\mathbf{0}}^{=}(\mathbf{0}) \equiv \mathsf{ind}_{\mathbf{0}}^{=}(x \xmapsto{\times} \mathbf{0}) : \mathbf{0} \xrightarrow{=} \mathbf{0}. \tag{2.111}$$

Inductive type 2.10.6. We define the **unit type 1** as the inductive type with the following constructor:

- $\star : \mathbf{1}$.

The recursion principle is:

$$\mathsf{rec}_{\mathbf{1}}^{v} : \prod_{C:\mathcal{U}_i}^{=} C \xrightarrow{+} (\mathbf{1} \xrightarrow{v} C), \tag{2.112}$$

with computation rule $\mathsf{rec}_{\mathbf{1}}^{v}(C, c)(\star) \equiv c : C$. The induction principle is:

$$\mathsf{ind}_{\mathbf{1}}^{v} : \prod_{C:\mathbf{1}\xrightarrow{\times}\mathcal{U}_i}^{=} C(\star) \xrightarrow{+} \left(\prod_{x:\mathbf{1}}^{v} C(x) \right), \tag{2.113}$$

with computation rule $\mathsf{ind}_{\mathbf{1}}^{v}(C, c)(\star) \equiv c : C(\star)$. We can define an isovariant identity function:

$$\mathrm{id}_{\mathbf{1}} :\equiv \mathsf{rec}_{\mathbf{1}}^{=}(\mathbf{1}, \star) : \mathbf{1} \xrightarrow{=} \mathbf{1}. \tag{2.114}$$

Definition 2.10.7. We define the function $\mathsf{Fin} : \mathbb{N} \xrightarrow{4} \mathcal{U}_0$, called the **family of finite sets**, recursively: $\mathsf{Fin}(0) :\equiv \mathbf{0}$ and $\mathsf{Fin}(\mathsf{succ}\, n) :\equiv \mathsf{Fin}(n) + \mathbf{1}$. Formally:

$$\mathsf{Fin} :\equiv \mathsf{rec}_{\mathbb{N}}(\mathcal{U}_0, \mathbf{0}, n \xmapsto{4} X \xmapsto{+} (X + \mathbf{1})). \tag{2.115}$$

The elements are still denoted as in symmetric HoTT (definition 2.10.3).

2.11 The product

This section is based on [Uni13, §1.5].

In symmetric HoTT

Inductive type 2.11.1. If A and B are types, we define their **product** $A \times B$ with the following constructor:

- $(\square, \square) : A \to B \to A \times B$.

The recursion principle is

$$\mathsf{rec}_{A\times B} : \prod_{C:\mathcal{U}_i} (A \to B \to C) \to (A \times B \to C) \tag{2.116}$$

with computation rule

$$\mathsf{rec}_{A\times B}(C, f)((a, b)) \equiv f(a)(b) : C. \tag{2.117}$$

The induction principle is, similarly,

$$\mathsf{ind}_{A\times B} : \prod_{C:A\times B\to\mathcal{U}_i} \left(\prod_{a:A} \prod_{b:B} C((a, b)) \right) \to \left(\prod_{x:A\times B} C(x) \right) \tag{2.118}$$

with computation rule

$$\mathsf{ind}_{A\times B}(C, f)((a, b)) \equiv f(a)(b) : C((a, b)). \tag{2.119}$$

Remark 2.11.2 (Currying)**.** If we have a function $A \times B \to C$, we can **curry** it to a function $A \to B \to C$:

$$\mathsf{curry} :\equiv f \mapsto (a \mapsto b \mapsto f((a, b))) : (A \times B \to C) \to (A \to B \to C). \tag{2.120}$$

The reverse is called **uncurrying**:

$$\mathsf{uncurry} :\equiv g \mapsto (\mathsf{rec}_{A\times B}(C, g)) : (A \to B \to C) \to (A \times B \to C). \tag{2.121}$$

They are each other's inverse:

$$\begin{aligned} \mathsf{uncurry}\,(\mathsf{curry}\, f)\,((a_0, b_0)) &\equiv \mathsf{rec}_{A\times B}(C, a \mapsto b \mapsto f((a, b)))((a_0, b_0)) \\ &\equiv (a \mapsto b \mapsto f((a, b)))\,(a_0)(b_0) \equiv f((a_0, b_0)), \qquad (2.122) \\ \mathsf{curry}\,(\mathsf{uncurry}\, g)\,(a_0)(b_0) &\equiv (a \mapsto b \mapsto \mathsf{rec}_{A\times B}(C, g)((a, b)))\,(a_0)(b_0) \\ &\equiv \mathsf{rec}_{A\times B}(C, g)((a_0, b_0)) \equiv g(a_0)(b_0). \qquad (2.123) \end{aligned}$$

We conclude that $(A \times B \to C) \simeq (A \to B \to C)$. We can also curry and uncurry dependent functions, yielding:

$$\left(\prod_{e:A\times B} C(e) \right) \simeq \left(\prod_{a:A} \prod_{b:B} C((a, b)) \right). \tag{2.124}$$

Remark 2.11.3 (Projections)**.** We define coordinate projections

$$\begin{aligned} \mathsf{prl} &:\equiv \mathsf{rec}_{A\times B}(A, a \mapsto b \mapsto a) : A \times B \to A, \\ \mathsf{prr} &:\equiv \mathsf{rec}_{A\times B}(B, a \mapsto b \mapsto b) : A \times B \to B. \end{aligned} \tag{2.125}$$

Then we have

$$\mathsf{prl}((a,b)) \equiv a : A, \qquad \mathsf{prr}((a,b)) \equiv b : B. \tag{2.126}$$

Assuming a uniqueness principle ($e \equiv (\mathsf{prl}\, e, \mathsf{prr}\, e)$), the coordinate projections are as strong as the induction principle. Indeed, we have[ab]

$$\prod_{e:A\times B} \mathsf{ind}_{A\times B}(C,f)(e) = f(\mathsf{prl}\, e)(\mathsf{prr}\, e). \tag{2.127}$$

If we want to prove this, the induction principle tells us that we need only prove:

$$\prod_{a:A}\prod_{b:B} \mathsf{ind}_{A\times B}(C,f)((a,b)) = f(\mathsf{prl}\,(a,b))(\mathsf{prr}\,(a,b)) \tag{2.128}$$

but both left and right hand side are judgementally equal to $f(a)(b)$, so we conclude this from reflexivity of $=$:

$$a \mapsto b \mapsto \mathsf{refl}(f(a)(b)) : \prod_{a:A}\prod_{b:B} f(a)(b) = f(a)(b). \tag{2.129}$$

[a] Remember that the product sign applies to everything on its right.

[b] The uniqueness principle is needed to even state this equation, as $\mathsf{ind}_{A\times B}(C,f)(e) : C(e)$ but $f(\mathsf{prl}\, e)(\mathsf{prr}\, e) : C(\mathsf{prl}\, e, \mathsf{prr}\, e)$.

In directed HoTT

Inductive type 2.11.4. If A and B are types, we define their **product** $A \times B$, which is covariant in both A and B, with the following constructor:

- $(\square, \square) : A \xrightarrow{+} B \xrightarrow{+} A \times B$.

The recursion principle is

$$\mathsf{rec}^v_{A\times B} : \overset{=}{\prod_{C:\mathcal{U}_i}} (A \xrightarrow{v} B \xrightarrow{v} C) \xrightarrow{+} (A \times B \xrightarrow{v} C). \tag{2.130}$$

with computation rule

$$\mathsf{rec}^v_{A\times B}(C,f)((a,b)) \equiv f(a)(b) : C. \tag{2.131}$$

The induction principle is,

$$\mathsf{ind}^v_{A\times B} : \overset{=}{\prod_{C:A\times B\to\mathcal{U}_i}} \left(\prod_{a:A}^{v}\prod_{b:B}^{v} C((a,b))\right) \xrightarrow{+} \left(\prod_{x:A\times B}^{v} C(x)\right) \tag{2.132}$$

with computation rule

$$\mathsf{ind}^v_{A\times B}(C,f)((a,b)) \equiv f(a)(b) : C((a,b)). \tag{2.133}$$

Remark 2.11.5 (Currying)**.** If we have a function $A \times B \xrightarrow{v} C$, we can **curry** it to a function $A \xrightarrow{v} B \xrightarrow{v} C$:

$$\mathsf{curry}^v :\equiv f \overset{+}{\mapsto} \left(a \overset{v}{\mapsto} b \overset{v}{\mapsto} f((a,b))\right) : (A \times B \xrightarrow{v} C) \xrightarrow{+} (A \xrightarrow{v} B \xrightarrow{v} C). \tag{2.134}$$

We can also **uncurry**:

$$\mathsf{uncurry}^v :\equiv g \overset{v}{\mapsto} \left(\mathsf{rec}^v_{A\times B}(C, g)\right) : (A \xrightarrow{v} B \xrightarrow{v} C) \xrightarrow{+} (A \times B \xrightarrow{v} C). \tag{2.135}$$

So we still have $(A \times B \to C) \overset{+}{\simeq} (A \to B \to C)$. As $\mathsf{rec}_{A\times B}(C)$ is isovariant in A, B and C, and $\mapsto$ is isovariant in the domain and the codomain, curry^v and $\mathsf{uncurry}^v$ (which really should be indexed with A, B and C) are isovariant in those three types. We can also curry and uncurry dependent functions, yielding:

$$\left(\prod_{e:A\times B}^{v} C(e)\right) \overset{+}{\simeq} \left(\prod_{a:A}^{v}\prod_{b:B}^{v} C((a,b))\right). \tag{2.136}$$

Remark 2.11.6 (Projections)**.** We define coordinate projections

$$\begin{aligned}\mathsf{prl} &:\equiv \mathsf{rec}^+_{A\times B}(A, a \overset{+}{\mapsto} b \overset{+}{\mapsto} a) : A \times B \xrightarrow{+} A,\\ \mathsf{prr} &:\equiv \mathsf{rec}^+_{A\times B}(B, a \overset{+}{\mapsto} b \overset{+}{\mapsto} b) : A \times B \xrightarrow{+} B.\end{aligned} \tag{2.137}$$

We still have

$$\mathsf{prl}((a,b)) \equiv a : A, \qquad \mathsf{prr}((a,b)) \equiv b : B. \tag{2.138}$$

Just like the recursion principle that defines them, the prl and prr are isovariant in A and B.

Assuming $e \equiv (\mathsf{prl}\, e, \mathsf{prr}\, e)$, the coordinate projections are still as strong as the induction principle. Indeed, we have

$$\prod_{e:A\times B}^{=} \mathsf{ind}^v_{A\times B}(C, f)(e) = f(\mathsf{prl}\, e)(\mathsf{prr}\, e). \tag{2.139}$$

(Don't mind the isovariance in e, it comes from the fact that refl is isovariant, which will be explained in section 2.15.) If we want to prove this, the induction principle tells us that we need only prove:

$$\prod_{a:A}^{=}\prod_{b:B}^{=} \mathsf{ind}^v_{A\times B}(C, f)((a,b)) = f(\mathsf{prl}\,(a,b))(\mathsf{prr}\,(a,b)) \tag{2.140}$$

but both left and right hand side are judgementally equal to $f(a)(b)$, so we conclude

this from reflexivity of $=$:

$$a \overset{=}{\mapsto} b \overset{=}{\mapsto} \mathsf{refl}(f(a)(b)) : \prod_{a:A}^{=} \prod_{b:B}^{=} f(a)(b) = f(a)(b). \tag{2.141}$$

2.12 Opposite, core and localization

The types defined in this section only apply to the directed case.

2.12.1 The opposite of a type

In category theory, every category has an opposite category, in which all morphisms are reversed. We define a type that plays a similar role:

Inductive type 2.12.1. Given a type A, we define its **opposite** A^{op}, which is invariant in A, with the following constructor:

- $\mathsf{flip} : A \overset{-}{\to} A^{\mathsf{op}}$.

It may be surprising that we define A^{op} invariant in A. For types with only covariant constructors, Guideline 2.7.4 stipulates that a type may be v-variant in an object x as long as the types of all arguments are v-variant in x. Here, the only constructor flip takes one argument of type A. Since the identity is covariant, we would expect A^{op} to be covariant in A. Indeed, given a function $f : A \overset{+}{\to} C$, we can define $f' : A^{\mathsf{op}} \overset{+}{\to} C^{\mathsf{op}}$ by $f'(\mathsf{flip}\, a) :\equiv \mathsf{flip}\, f(a) : C^{\mathsf{op}}$. However, the mapping $f \overset{-}{\mapsto} f'$ is contravariant. Indeed, assuming directed function extensionality (see section 3.8.8), a morphism $f \rightsquigarrow g$ is a pointwise morphism $\prod_{a:A}^{=} f(a) \rightsquigarrow_C g(a)$. But this yields a pointwise morphism $\prod_{a:A}^{=} \mathsf{flip}\, g(a) \rightsquigarrow_{C^{\mathsf{op}}} \mathsf{flip}\, f(a)$, i.e. a morphism $g' \rightsquigarrow f'$.

If we want maximal power, we would have to define $\Box^{\mathsf{op}}$ as a function that preserves morphisms contravariantly. However, as this variance is not natively available in the theory, we have to settle for invariance.

In general, if a type $A[x]$ depends on an object $x : X$ and a constructor takes an argument of type $T[x]$ v-variantly, where $T[x]$ is t-variant in x, then from a morphism $x \rightsquigarrow_X y$ we get a bridge/morphism/equality (depending on t) between $T[x]$ and $T[y]$, which we can use v-variantly to obtain a bridge/morphism/equality between $A[x]$ and $A[y]$. So the variance of $A[x]$ is not stronger than $(+_0 v_1) \circ t$, where $(+_0 v_1)$-variant functions preserve bridges/morphisms/equalities v-variantly. As the variance $(+_0 v_1)$ is not available natively, we have to weaken it. If $v \in \{+, =\}$, then we can weaken it to $+$, otherwise we weaken it to $\times$. We extract a new guideline from this, generalizing 2.7.4 on page 32:

Guideline 2.12.2. If a type $A[x]$ depends on an object $x : X$ and a constructor takes an argument of type $T[x]$ v-variantly, where $T[x]$ is t-variant in x, then the variance of $A[x]$ must be weaker than:

- t, if $v \in \{+, =\}$,
- $\times \circ t$, if $v \in \{-, \times\}$.

We do not consider the variance of inductive types with constructors that take recursive arguments in a non-covariant way, such as $:: : A \xrightarrow{+} \mathsf{Fliplist}\, A \xrightarrow{-} \mathsf{Fliplist}\, A$.

To create a function $A^{\mathsf{op}} \xrightarrow{v} C$, we need to give a function $A \xrightarrow{v \circ -} C$. Thus, as prescribed by Guideline 2.7.5, the recursion principle is:

$$\mathsf{rec}^v_{A^{\mathsf{op}}} : \prod_{C:\mathcal{U}_i}^{=} \left(A \xrightarrow{v\circ -} C\right) \xrightarrow{+} \left(A^{\mathsf{op}} \xrightarrow{v} C\right), \tag{2.142}$$

with computation rule: $\mathsf{rec}^v_{A^{\mathsf{op}}}(C, f)(\mathsf{flip}\, x) \equiv f(x) : C$. The induction principle is:

$$\mathsf{ind}^v_{A^{\mathsf{op}}} : \prod_{C:A^{\mathsf{op}} \xrightarrow{\times} \mathcal{U}_i}^{=} \left(\prod_{a:A}^{v\circ -} C(\mathsf{flip}\, a)\right) \xrightarrow{+} \left(\prod_{a':A^{\mathsf{op}}}^{v} C(a')\right), \tag{2.143}$$

with computation rule: $\mathsf{ind}^v_{A^{\mathsf{op}}}(C, f)(\mathsf{flip}\, x) \equiv f(x) : C(\mathsf{flip}\, x)$.

Remark 2.12.3 (Opposite currying)**.** There is an analogue of currying for the opposite type, which we will call **opposite currying**. (This is typical for inductive types with only a single constructor.) Opposite currying is not the opposite of currying, but rather currying for the opposite type.

If we have a function $A^{\mathsf{op}} \xrightarrow{v} C$, we can **curry** it to a function $A \xrightarrow{v\circ -} C$:

$$\mathsf{opCurry}^v :\equiv f \xmapsto{+} \left(a \xmapsto{v\circ -} f(\mathsf{flip}\, a)\right) : \left(A^{\mathsf{op}} \xrightarrow{v} C\right) \xrightarrow{+} \left(A \xrightarrow{v\circ -} C\right). \tag{2.144}$$

We can also **uncurry**:

$$\mathsf{opUncurry}^v :\equiv g \xmapsto{+} (\mathsf{rec}^v_{A^{\mathsf{op}}}(C, g)) : \left(A \xrightarrow{v\circ -} C\right) \xrightarrow{+} \left(A^{\mathsf{op}} \xrightarrow{v} C\right). \tag{2.145}$$

The currying and uncurrying functions are isovariant in the relevant types. They are each other's inverse:

$$\begin{aligned} \mathsf{opUncurry}^v\, (\mathsf{opCurry}^v\, f)\, (\mathsf{flip}\, a_0) &\equiv \mathsf{rec}_{A^{\mathsf{op}}}(C, a \xmapsto{v\circ -} f(\mathsf{flip}\, a))(\mathsf{flip}\, a_0) \\ &\equiv \left(a \xmapsto{v\circ -} f(\mathsf{flip}\, a)\right)(a_0) \equiv f(\mathsf{flip}\, a_0), \qquad (2.146) \\ \mathsf{opCurry}^v\, (\mathsf{opUncurry}^v\, g)\, (a_0) &\equiv \left(a \xmapsto{v\circ -} \mathsf{rec}_{A^{\mathsf{op}}}(C, g)(\mathsf{flip}\, a)\right)(a_0) \\ &\equiv \mathsf{rec}_{A^{\mathsf{op}}}(C, g)(\mathsf{flip}\, a_0) \equiv g(a_0). \qquad (2.147) \end{aligned}$$

We conclude that $\left(A^{\mathsf{op}} \xrightarrow{v} C\right) \overset{+}{\simeq} \left(A \xrightarrow{v \circ -} C\right)$. We can also curry and uncurry dependent functions, yielding:

$$\left(\prod_{e:A^{\mathsf{op}}}^{v} C(e)\right) \overset{+}{\simeq} \left(\prod_{a:A}^{v \circ -} C(\mathsf{flip}\, a)\right). \tag{2.148}$$

Remark 2.12.4 (Unflipping)**.** There is also an analogue of the projections from the product. (Again, this is typical for inductive types with only one constructor.) Whereas projections can be seen as the inverses of pairing, here we will get an inverse of flip. We will aptly call it unflip:

$$\mathsf{unflip} :\equiv \mathsf{rec}^{-}_{A^{\mathsf{op}}}(A, a \overset{+}{\mapsto} a) : A^{\mathsf{op}} \xrightarrow{-} A. \tag{2.149}$$

Then we have $\mathsf{unflip}\,(\mathsf{flip}\, a) \equiv a : A$.

The unflipping function is isovariant in the type A. Assuming a uniqueness principle ($e \equiv \mathsf{flip}\,(\mathsf{unflip}\, e)$), it is as strong as the induction principle. Indeed, we have

$$\prod_{e:A^{\mathsf{op}}}^{=} \mathsf{ind}^{v}_{A^{\mathsf{op}}}(C, f)(e) = f(\mathsf{unflip}\, e). \tag{2.150}$$

If we want to prove this, the induction principle tells us that we need only prove:

$$\prod_{a:A}^{=} \mathsf{ind}^{v}_{A^{\mathsf{op}}}(C, f)(\mathsf{flip}\, a) = f(\mathsf{unflip}\,(\mathsf{flip}\, a)) \tag{2.151}$$

but both left and right hand side are judgementally equal to $f(a)$, so we conclude this from reflexivity of $=$:

$$a \overset{=}{\mapsto} \mathsf{refl}(f(a)) : \prod_{a:A}^{=} f(a) = f(a). \tag{2.152}$$

One can prove, additionally, that $\mathsf{flip}\,(\mathsf{unflip}\, a) = a$. We conclude that A and A^{op} are contravariantly equivalent (a notion which we will not define formally): $A \overset{-}{\simeq} A^{\mathsf{op}}$. Then A is covariantly equivalent to $A^{\mathsf{op}\mathsf{op}}$: $A \overset{+}{\simeq} A^{\mathsf{op}\mathsf{op}}$, and $\mathsf{flip} : A^{\mathsf{op}} \xrightarrow{-} A^{\mathsf{op}\mathsf{op}}$ corresponds to $\mathsf{unflip} : A^{\mathsf{op}} \xrightarrow{-} A$ under this equivalence.

From the contravariant equivalence, we also see (by composing) that a function $A \xrightarrow{v} C$ is the same as a function $A \xrightarrow{-\circ v} C^{\mathsf{op}}$. This equivalence, too, has a variance, which follows from the variance of the composition law (see definition 2.4.3):

$$\left(A \xrightarrow{v} C\right) \overset{-}{\simeq} A \xrightarrow{-\circ v} C^{\mathsf{op}}. \tag{2.153}$$

2.12.2 The core of a type

In category theory

Every category has an underlying objects set (or class), which consists of the category's objects, but not its morphisms. We want to do a similar thing in type theory. However, as we want isomorphism to coincide with equality, and everything should respect equality, we cannot throw away isomorphisms. Thus, the best thing we can do is to take the core of a category.

Definition 2.12.5. The **core** of a category $\mathcal{C}$ is the category $\mathcal{C}^{\mathsf{core}}$ whose objects are the objects of $\mathcal{C}$ and whose morphisms are the isomorphisms of $\mathcal{C}$ [nLa12]. We have a functor $E : \mathcal{C}^{\mathsf{core}} \to \mathcal{C}$ that maps objects and morphisms to themselves.

The core obeys a universal property:

Proposition 2.12.6. If $\mathcal{G}$ is a groupoid and $F : \mathcal{G} \to \mathcal{C}$ is a functor, then there exists a unique $F' : \mathcal{G} \to \mathcal{C}^{\mathsf{core}}$ so that $F = E \circ F'$.

Proof. Existence: Let φ be any morphism in $\mathcal{G}$. Since $\mathcal{G}$ is a groupoid, φ is an isomorphism, and then so is $F\varphi$. But then $F\varphi$ is a morphism in $\mathcal{C}^{\mathsf{core}}$, so we can define $F' := F$. Then clearly $F = E \circ F'$.

Uniqueness: This follows from the fact that E is injective on the objects class as well as on every Hom-set. □

Remark 2.12.7. When we consider $\square^{\mathsf{core}} : \mathsf{Cat} \to \mathsf{Grpd}$ as a 1-functor between 2-categories (preserving morphisms (functors in Cat) but not morphisms between morphisms (natural transformations in Cat)), then we can also state the universal property by saying that $\square^{\mathsf{core}}$ is right adjoint to the forgetful functor $U : \mathsf{Grpd} \to \mathsf{Cat}$ [nLa12].

In directed HoTT

We define an analogous type:

Inductive type 2.12.8. Given a type A, we define its **core** A^{core}, which is invariant in A, with the following constructor:

- $\mathsf{strip} : A \xrightarrow{\times} A^{\mathsf{core}}$.

Again, the function $\square^{\mathsf{core}}$ does preserve morphisms, but it preserves them invariantly. The strongest basic variance that is consistent with this behaviour, is invariance. To create a function $A^{\mathsf{core}} \xrightarrow{v} C$, we need to give a function $A \xrightarrow{v \circ \times} C$:

$$\mathsf{rec}^{v}_{A^{\mathsf{core}}} : \prod_{C:\mathcal{U}_i}^{=} \left(A \xrightarrow{v\circ\times} C\right) \xrightarrow{+} \left(A^{\mathsf{core}} \xrightarrow{v} C\right). \tag{2.154}$$

The computation rule is: $\mathsf{rec}^{v}_{A^{\mathsf{core}}}(C, f)(\mathsf{strip}\, x) \equiv f(x) : C$.

The induction principle is:

$$\mathsf{ind}^{v}_{A^{\mathsf{core}}} : \prod^{=}_{C:A^{\mathsf{core}} \overset{\times}{\to} \mathcal{U}_i} \left(\prod_{a:A}^{v \circ \times} C(\mathsf{strip}\, a) \right) \overset{+}{\to} \left(\prod_{a':A^{\mathsf{core}}}^{v} C(a') \right), \tag{2.155}$$

with computation rule: $\mathsf{ind}^{v}_{A^{\mathsf{core}}}(C, f)(\mathsf{strip}\, x) \equiv f(x) : C(\mathsf{strip}\, x)$.

Remark 2.12.9 (Core currying)**.** Again, there is an analogue of currying. If we have a function $A^{\mathsf{core}} \overset{v}{\to} C$, we can **core curry** it to a function $A \overset{v \circ \times}{\to} C$:

$$\mathsf{coreCurry}^{v} :\equiv f \overset{+}{\mapsto} \left(a \overset{v \circ \times}{\mapsto} f(\mathsf{strip}\, a) \right) : \left(A^{\mathsf{core}} \overset{v}{\to} C \right) \overset{+}{\to} \left(A \overset{v \circ \times}{\to} C \right). \tag{2.156}$$

We can also **uncurry**:

$$\mathsf{coreUncurry}^{v} :\equiv g \overset{+}{\mapsto} (\mathsf{rec}^{v}_{A^{\mathsf{core}}}(C, g)) : \left(A \overset{v \circ \times}{\to} C \right) \overset{+}{\to} \left(A^{\mathsf{core}} \overset{v}{\to} C \right). \tag{2.157}$$

The currying and uncurrying functions are isovariant in the relevant types. They are each other's inverse:

$$\begin{aligned} \mathsf{coreUncurry}^{v}\, (\mathsf{coreCurry}^{v}\, f)\, (\mathsf{strip}\, a_0) &\equiv \mathsf{rec}_{A^{\mathsf{core}}}(C, a \overset{v \circ \times}{\mapsto} f(\mathsf{strip}\, a))(\mathsf{strip}\, a_0) \\ &\equiv \left(a \overset{v \circ \times}{\mapsto} f(\mathsf{strip}\, a) \right)(a_0) \equiv f(\mathsf{strip}\, a_0), \qquad (2.158) \\ \mathsf{coreCurry}^{v}\, (\mathsf{coreUncurry}^{v}\, g)\, (a_0) &\equiv \left(a \overset{v \circ \times}{\mapsto} \mathsf{rec}_{A^{\mathsf{core}}}(C, g)(\mathsf{strip}\, a) \right)(a_0) \\ &\equiv \mathsf{rec}_{A^{\mathsf{core}}}(C, g)(\mathsf{strip}\, a_0) \equiv g(a_0). \qquad (2.159) \end{aligned}$$

We conclude that $\left(A^{\mathsf{core}} \overset{v}{\to} C \right) \overset{+}{\simeq} \left(A \overset{v \circ \times}{\to} C \right)$. We can also curry and uncurry dependent functions, yielding:

$$\left(\prod_{e:A^{\mathsf{core}}}^{v} C(e) \right) \overset{+}{\simeq} \left(\prod_{a:A}^{v \circ \times} C(\mathsf{strip}\, a) \right). \tag{2.160}$$

Remark 2.12.10 (Unstripping)**.** There is also an analogue of the projections from the product and unflipping for the opposite:

$$\mathsf{unstrip} :\equiv \mathsf{rec}^{\pm}_{A^{\mathsf{core}}}(A, a \overset{\times}{\to} a) : A^{\mathsf{core}} \overset{\pm}{\to} A. \tag{2.161}$$

Then we have $\mathsf{unstrip}\, (\mathsf{strip}\, a) \equiv a : A$. The recursion principle allows us to define $\mathsf{unstrip}$ co-, contra- or invariantly, but not isovariantly. Unfortunately, these three variances have no 'maximum', so we have to make do with the $\pm$ notation.

The unstripping function is isovariant in A. Assuming a uniqueness principle ($e :^{\times} A^{\mathsf{core}} \vdash e \equiv \mathsf{strip}\, (\mathsf{unstrip}\, e) :^{+} A^{\mathsf{core}}$), it is as strong as the induction principle, but it is less straightforward to see it. For the invariant and isovariant induction

principle, we can get the variances straight right away:

$$\prod_{e:A^{\mathsf{core}}}^{=} \mathsf{ind}^{=}_{A^{\mathsf{core}}}(C, f)(e) = f(\mathsf{unstrip}\, e),$$
$$\prod_{e:A^{\mathsf{core}}}^{=} \mathsf{ind}^{\times}_{A^{\mathsf{core}}}(C, f)(e) = f(\mathsf{unstrip}\, e). \tag{2.162}$$

In the first case, the left hand side has variance $=$ in e and the right hand side has $= \circ \times \circ \pm$ (that is $= \circ \times$ for f and $\pm$ for $\mathsf{unstrip}$), which match. In the second case, we get $\times$ on the left, and $\times \circ \times \circ \pm$ on the right, which also match. But for the covariant principle, we would get $+$ on the left and $+ \circ \times \circ \pm$ on the right, which is weaker. So we would be weakening the covariant induction principle to an invariant one. However, from (2.160), we can conclude:

$$\left(\prod_{a:A}^{\times} C(\mathsf{strip}\, a)\right) \overset{+}{\simeq} \left(\prod_{e:A^{\mathsf{core}}}^{+} C(e)\right) \overset{+}{\simeq} \left(\prod_{e:A^{\mathsf{core}}}^{-} C(e)\right) \overset{+}{\simeq} \left(\prod_{e:A^{\mathsf{core}}}^{\times} C(e)\right), \tag{2.163}$$

so that we only need the invariant and isovariant induction principles.

The proof of (2.162) is analogous to the one for unflip.

Equation (2.163) is not surprising: co-, contra- and invariant functions act the same way on bridges and isomorphisms; they only act differently on non-invertible morphisms, which we have thrown away in A^{core}. The core may however contain non-trivial bridges, so we cannot add $A^{\mathsf{core}} \overset{=}{\to} C$ to this sequence and in particular, we cannot construct an isovariant identity function from A^{core} to A^{core}. Indeed, to use the recursion principle to define such a function, we would need an $(= \circ \times) \equiv =$-variant function $A \overset{=}{\to} A^{\mathsf{core}}$.

2.12.3 Bridged groupoids

We will call a type such as A^{core}, with non-trivial bridges but all trivial morphisms, a bridged groupoid. The best way to express that a type has this property, is by saying that it remains covariantly equivalent when you remove all non-trivial morphisms but not the bridges. The function that does this, is strip, and we already know that this function has a covariant inverse $\mathsf{unstrip}$, so the only extra thing we need is that $\mathsf{unstrip}$ be an equivalence:

Definition 2.12.11. A type A is a **bridged groupoid** if the function $\mathsf{unstrip} : A^{\mathsf{core}} \overset{+}{\to} A^{\mathsf{core}}$ is an equivalence:

$$\mathsf{isBridgedGrpd}(A) :\equiv \mathsf{isEquiv}(\mathsf{unstrip}). \tag{2.164}$$

See section section 3.3 for a definition of $\mathsf{isEquiv}$. As the predicate $\mathsf{isEquiv}(f)$ is invariant in the domain and codomain of f, the predicate $\mathsf{isBridgedGrpd}(A)$ is invariant in A.

We illustrate that A^{core} is always a bridged groupoid. Define:

$$f :\equiv \mathsf{rec}^{+}_{A^{\mathsf{core}}}(A^{\mathsf{corecore}}, a \overset{\times}{\mapsto} \mathsf{strip}\,(\mathsf{strip}\,a)) : A^{\mathsf{core}} \overset{+}{\to} A^{\mathsf{corecore}}. \tag{2.165}$$

It computes: $f(\mathsf{strip}\,e) \equiv \mathsf{strip}\,\mathsf{strip}\,e$. We show that it is the inverse of $\mathsf{unstrip} : A^{\mathsf{corecore}} \overset{+}{\to} A^{\mathsf{core}}$. Take $a' :^{+} A^{\mathsf{core}}$. By induction, we may assume that $a' \equiv \mathsf{strip}\,a$ for some $a :^{\times} A$. Then

$$\mathsf{unstrip}\,f(a') \equiv \mathsf{unstrip}\,f(\mathsf{strip}\,a) \equiv \mathsf{unstrip}\,(\mathsf{strip}\,(\mathsf{strip}\,a)) \equiv \mathsf{strip}\,a \equiv a'. \tag{2.166}$$

Conversely, take $a'' :^{+} A^{\mathsf{corecore}}$. By double induction, we may assume that $a'' \equiv \mathsf{strip}\,(\mathsf{strip}\,a)$ for some $a :^{\times} A$. Then

$$f(\mathsf{unstrip}\,a'') \equiv f(\mathsf{unstrip}\,(\mathsf{strip}\,(\mathsf{strip}\,a))) \equiv f(\mathsf{strip}\,a) \equiv \mathsf{strip}\,(\mathsf{strip}\,a) \equiv a''. \tag{2.167}$$

Now, if G is a bridged groupoid, then $G \overset{+}{\simeq} G^{\mathsf{core}}$, so we can translate (2.163) to

$$\left(\prod_{x:G}^{+} C(x)\right) \overset{+}{\simeq} \left(\prod_{x:G}^{-} C(x)\right) \overset{+}{\simeq} \left(\prod_{x:G}^{\times} C(x)\right) \tag{2.168}$$

by composing with that equivalence. I.e. for functions from a bridged groupoid, it is irrelevant which one of these three variances they have. We will use 3 as a variance annotation to remind of this irrelevance.

There is a type theoretic analogue of the universal property for the core: if G is a bridged groupoid and $f : G \overset{3}{\to} A$, then we construct a function $f' : G \overset{3}{\to} A^{\mathsf{core}}$ so that $f(x) =_A \mathsf{unstrip}\,f'(x)$ for all $x : G$:

$$f' :\equiv x \overset{3}{\mapsto} \mathsf{strip}\,f(x) : G \overset{3}{\to} A^{\mathsf{core}} \tag{2.169}$$

This is clearly the only solution, as $\mathsf{unstrip}$ is invertible.

2.12.4 The localization of a type

In category theory

At this point, for any type A we have a type A^{op} so that a contravariant function from A is the same a covariant one from A^{op}, and a type A^{core} so that an invariant function from A is the same as a covariant one from A^{core}. We would like a similar thing for isovariance. For that, we will use the concept of localization from category theory.

Definition 2.12.12. Let $\mathcal{C}$ be a category and W a set of morphisms in $\mathcal{C}$. A **localization of $\mathcal{C}$ away from** W consists of a category $\mathcal{C}[W^{-1}]$ and a functor $P_W : \mathcal{C} \to \mathcal{C}[W^{-1}]$ with the following universal property:

Whenever $F : \mathcal{C} \to \mathcal{D}$ is a functor that maps any morphism in W to an isomorphism, then there exists a unique functor $F' : \mathcal{C}[W^{-1}] \to \mathcal{D}$ so that $F = F' \circ P_W$. [GZ67]

What we want to do is turn every morphism into an equality, i.e. an isomorphism. Thus, we will be using the localization away from *all* morphisms, which we will simply call *the* localization. For *the* localization, we can also state the universal property by saying that the localization 1-functor $\Box^{\mathsf{loc}} : \mathsf{Cat} \to \mathsf{Grpd}$ is left adjoint to the forgetful 1-functor $U : \mathsf{Grpd} \to \mathsf{Cat}$ [nLa12].

In directed HoTT

Inductive type 2.12.13. Given a type A, we define its **localization** A^{loc}, which is covariant in A, with the following constructor:

- $\mathsf{gather} : A \overset{=}{\to} A^{\mathsf{loc}}$.

This time, A^{loc} is covariant in A. In fact, it has stronger variance: it preserves morphisms, but it preserves them isovariantly.

Note that for any variance v, $v \circ = \equiv =$. So, to create a function $A^{\mathsf{loc}} \overset{v}{\to} C$, we need to give a function $A \overset{=}{\to} C$:

$$\mathsf{rec}^{v}_{A^{\mathsf{loc}}} : \prod_{C:\mathcal{U}_i}^{=} \left(A \overset{=}{\to} C\right) \overset{+}{\to} \left(A^{\mathsf{loc}} \overset{v}{\to} C\right). \tag{2.170}$$

The computation rule is: $\mathsf{rec}^{v}_{A^{\mathsf{loc}}}(C, f)(\mathsf{gather}\, x) \equiv f(x) : C$.

The induction principle is:

$$\mathsf{ind}^{v}_{A^{\mathsf{loc}}} : \prod_{C:A^{\mathsf{loc}} \overset{\times}{\to} \mathcal{U}_i}^{=} \left(\prod_{a:A}^{\times} C(\mathsf{gather}\, a)\right) \overset{+}{\to} \left(\prod_{a':A^{\mathsf{loc}}}^{v} C(a')\right), \tag{2.171}$$

with computation rule: $\mathsf{ind}^{v}_{A^{\mathsf{loc}}}(C, f)(\mathsf{gather}\, x) \equiv f(x) : C(\mathsf{gather}\, x)$.

Remark 2.12.14 (Localization currying)**.** If we have a function $A^{\mathsf{loc}} \overset{v}{\to} C$, we can **localization curry** it to a function $A \overset{=}{\to} C$:

$$\mathsf{locCurry}^{v} :\equiv f \overset{+}{\mapsto} \left(a \overset{=}{\mapsto} f(\mathsf{gather}\, a)\right) : \left(A^{\mathsf{loc}} \overset{v}{\to} C\right) \overset{+}{\to} \left(A \overset{=}{\to} C\right). \tag{2.172}$$

We can also **uncurry**:

$$\mathsf{locUncurry}^{v} :\equiv g \overset{+}{\mapsto} (\mathsf{rec}^{v}_{A^{\mathsf{core}}}(C, g)) : \left(A \overset{=}{\to} C\right) \overset{+}{\to} \left(A^{\mathsf{loc}} \overset{v}{\to} C\right). \tag{2.173}$$

The currying and uncurrying functions are isovariant in the relevant types. They are each other's inverse:

$$\begin{aligned} \mathsf{locUncurry}^{v}\,(\mathsf{locCurry}^{v}\, f)\,(\mathsf{gather}\, a_0) &\equiv \mathsf{rec}_{A^{\mathsf{loc}}}(C, a \overset{=}{\mapsto} f(\mathsf{gather}\, a))(\mathsf{gather}\, a_0) \\ &\equiv \left(a \overset{=}{\mapsto} f(\mathsf{gather}\, a)\right)(a_0) \equiv f(\mathsf{gather}\, a_0), \end{aligned} \tag{2.174}$$

$$\begin{aligned}\mathsf{locCurry}^v\,(\mathsf{locUncurry}^v\,g)\,(a_0) &\equiv \left(a \overset{=}{\mapsto} \mathsf{rec}_{A^{\mathsf{loc}}}(C,g)(\mathsf{gather}\,a)\right)(a_0)\\ &\equiv \mathsf{rec}_{A^{\mathsf{loc}}}(C,g)(\mathsf{gather}\,a_0) \equiv g(a_0). \end{aligned} \tag{2.175}$$

We conclude that $\left(A^{\mathsf{loc}} \overset{v}{\to} C\right) \overset{+}{\simeq} \left(A \overset{=}{\to} C\right)$. We can also curry and uncurry dependent functions, yielding:

$$\left(\prod_{e:A^{\mathsf{loc}}}^{v} C(e)\right) \overset{+}{\simeq} \left(\prod_{a:A}^{=} C(\mathsf{gather}\,a)\right). \tag{2.176}$$

Unlike flip and strip, gather has no inverse. This is because A^{loc} has more equalities than A, and any function, even an invariant one, must preserve equality. When defining flip and strip, we fed the identity function to the recursion principle. However, $\mathsf{rec}^v_{A^{\mathsf{loc}}}$ only takes isovariant functions.

From (2.176), we get:

$$\left(\prod_{e:A^{\mathsf{loc}}}^{+} C(e)\right) \overset{+}{\simeq} \left(\prod_{e:A^{\mathsf{loc}}}^{-} C(e)\right) \overset{+}{\simeq} \left(\prod_{e:A^{\mathsf{loc}}}^{\times} C(e)\right) \overset{+}{\simeq} \left(\prod_{e:A^{\mathsf{loc}}}^{=} C(e)\right) \tag{2.177}$$

This is not surprising: we have turned all structure into isomorphisms, and all functions preserve isomorphisms, so variance is immaterial for functions from A^{loc}. We can also construct an isovariant identity function $\mathrm{id}^{=}_{A^{\mathsf{loc}}}$:

$$\mathrm{id}^{=}_{A^{\mathsf{loc}}} :\equiv \mathsf{rec}^{=}_{A^{\mathsf{loc}}}(C, \mathsf{gather}). \tag{2.178}$$

2.12.5 Bridgeless groupoids

A bridgeless groupoid is a type that has only trivial bridges and morphisms: they are actually equalities. We can express this by saying that the identity function turns bridges and morphisms into equalities, i.e. that it is isovariant.

Definition 2.12.15. A type A is a **bridgeless groupoid** if the identity function id_A is isovariant:

$$\mathsf{isBridgelessGrpd}(A) :\equiv \sum_{f:A\overset{=}{\to}A}^{+} (f_+ = \mathrm{id}_A), \tag{2.179}$$

where f_+ is the function f seen as a covariant function: $f_+ :\equiv a \overset{+}{\mapsto} f(a)$. As the type $A \overset{+}{\to} A$ is invariant in A, the entire predicate $\mathsf{isBridgelessGrpd}(A)$ is invariant in A.

By composing with the identity function, we find that all functions to and from a bridgeless groupoid are isovariant and hence have any variance. As mentioned before, we will use the variance annotation 4 to remind that a function's variance is irrelevant.

Note that the situation is different for bridged and bridgeless groupoids. If G

is a bridged groupoid, we can write $G \xrightarrow{3} C$ but not $C \xrightarrow{3} G$ as functions $C \xrightarrow{\pm} G$ take morphisms to morphisms (therefore to isomorphisms) whereas $C \xrightarrow{\times} G$ takes morphisms to bridges. If H is a bridgeless groupoid, we can write both $H \xrightarrow{4} C$ and $C \xrightarrow{4} H$.

There is another way to state the universal property of the localization: if H is a bridgeless groupoid and $f : A \xrightarrow{4} H$, then we construct a function $f' : A^{\mathsf{loc}} \xrightarrow{4} H$ so that $f = f' \circ \mathsf{gather}$:

$$f' :\equiv \mathsf{rec}^{v}_{A^{\mathsf{loc}}}(H, f). \tag{2.180}$$

The induction principle allows us to prove that this f' is the only solution.

2.13 Dependent pair types

This section is based on [Uni13, §1.6].

In symmetric HoTT

The product can be generalized to a dependent pair type $\sum_{x:A} B(x)$. The elements of this type are pairs of the form (a, b) where $a : A$ and $b : B(a)$. So the type of the second component depends on the value of the first component. It is denoted with a summation sign because it may be regarded as the sum/coproduct/disjoint union of all types $B(a)$. For example, the type of all X-vectors of any length, could be defined as $\sum_{n:\mathbb{N}} \mathsf{Vec}_n X$.

Inductive type 2.13.1. If A is a type and $B[x]$ a type depending[a] on a variable $x : A$, we define $\sum_{a:A} B[a]$ as the inductive type with the following constructor:

- $(\square, \square) : \prod_{x:A} B[x] \to \sum_{a:A} B[a]$.

Remember that the product sign applies to everything on its right.

[a] We might as well have required a function $B : A \to \mathcal{U}_i$. Thanks to the inference rules of dependent functions, the result would have been equivalent. However, since we could not do this when defining dependent functions on penalty of a circular definition, we do not do it here either for conformity. It does not matter really.

The type of the second argument of our constructor, depends on the value of the first one. We celebrate this small novelty with a guideline:

Guideline 2.13.2. The type of a constructor's argument, may depend on the values of all preceding arguments. The recursion and induction principles become dependent accordingly.

Let us abbreviate $\sum_{x:A} B[x]$ as S. The recursion principle should tell us that, to create a function $S \to C$, we need a function $\prod_{x:A} B[x] \to C$. Thus, the recursion principle is:

$$\mathsf{rec}_S : \prod_{C:\mathcal{U}_i} \left(\prod_{a:A} B[a] \to C \right) \to (S \to C)\,, \tag{2.181}$$

with computation rule: $\mathsf{rec}_S(C, f)((a, b)) \equiv f(a)(b) : C$.

To create a function $\prod_{e:S} C(e)$, we need a function $\prod_{a:A} \prod_{b:B[a]} C((a,b))$. thus, the induction principle is:

$$\mathsf{ind}_S : \prod_{C:S\to\mathcal{U}_i} \left(\prod_{a:A} \prod_{b:B[a]} C((a,b)) \right) \to \left(\prod_{e:S} C(e) \right), \tag{2.182}$$

with computation rule: $\mathsf{ind}_S(C,f)((a,b)) \equiv f(a)(b) : C((a,b))$.

Remark 2.13.3 (Currying)**.** If we have a function $\prod_{e:S} C(e)$, we can **curry** it to a function $\prod_{a:A} \prod_{b:B[a]} C((a,b))$:

$$\mathsf{curry} :\equiv f \mapsto (a \mapsto b \mapsto f((a,b))) : \left(\prod_{e:S} C(e) \right) \to \left(\prod_{a:A} \prod_{b:B[a]} C((a,b)) \right). \tag{2.183}$$

We can also **uncurry**:

$$\mathsf{uncurry} :\equiv g \mapsto (\mathsf{ind}_S(C,g)) : \left(\prod_{a:A} \prod_{b:B[a]} C((a,b)) \right) \to \left(\prod_{e:S} C(e) \right). \tag{2.184}$$

So we get

$$\left(\prod_{e:S} C(e) \right) \simeq \left(\prod_{a:A} \prod_{b:B[a]} C((a,b)) \right). \tag{2.185}$$

Remark 2.13.4 (Projections)**.** We define coordinate projections

$$\begin{aligned} \mathsf{prl} &:\equiv \mathsf{rec}_S(A, a \mapsto b \mapsto a) : S \to A, \\ \mathsf{prr} &:\equiv \mathsf{ind}_S(e \mapsto B[\mathsf{prl}\, e], a \mapsto b \mapsto b) : \prod_{e:S} B[e]. \end{aligned} \tag{2.186}$$

It is not entirely trivial that the definition of prr is legal. Note that $\mathsf{ind}_{A\times B}(e \mapsto B[\mathsf{prl}\, e])$ requires us to provide a function of type $\prod_{a:A} \prod_{b:B[a]} B[\mathsf{prl}(a,b)]$. But

$$\mathsf{prl}(a,b) \equiv (a' \mapsto b' \mapsto a')\,(a)(b) \equiv a : A. \tag{2.187}$$

Since b has type $B[a]$, the given function has the correct type.

We have

$$\mathsf{prl}((a,b)) \equiv a : A, \qquad \mathsf{prr}((a,b)) \equiv b : B[a]. \tag{2.188}$$

Assuming $e \equiv (\mathsf{prl}\, e, \mathsf{prr}\, e)$, the coordinate projections are as strong as the induction principle. Indeed, we have

$$\prod_{e:S} \mathsf{ind}_S(C,f)(e) = f(\mathsf{prl}\, e)(\mathsf{prr}\, e). \tag{2.189}$$

If we want to prove this, the induction principle tells us that we need only prove:

$$\prod_{a:A}\prod_{b:B[a]} \mathsf{ind}_S(C,f)((a,b)) = f(\mathsf{prl}\,(a,b))(\mathsf{prr}\,(a,b)) \tag{2.190}$$

but both left and right hand side are judgementally equal to $f(a)(b)$, so we conclude this from reflexivity of $=$:

$$a \mapsto b \mapsto \mathsf{refl}(f(a)(b)) : \prod_{a:A}\prod_{b:B[a]} f(a)(b) = f(a)(b). \tag{2.191}$$

In directed HoTT

Inductive type 2.13.5. If A is a type and $B[x]$ a type, depending on a variable $x :^{\times} A$ invariantly, we define the type of **u-variant dependent pairs** $\sum^u_{a:A} B[a]$ as the inductive type with the following constructor:

- $(\square \overset{u}{,} \square) : \prod^u_{x:A} B[x] \overset{+}{\to} \sum^u_{a:A} B[a]$.

The type $\sum^u_{a:A} B[a]$ is covariant in the type family $B[a]$. If $u \in \{+,=\}$, it is covariant in A, otherwise it is invariant in A, as per Guideline 2.12.2 on page 45.

Let us abbreviate $\sum^u_{a:A} B[a]$ as S^u. Note that a pair $(a \overset{u}{,} b)$ is u-variant in a and covariant in b. The recursion principle for creating v-variant functions of pairs (which are thus $(v \circ u)$-variant in the first component), is:

$$\mathsf{rec}^v_{S^u} : \overset{=}{\prod_{C:\mathcal{U}_i}} \left(\prod_{a:A}^{v\circ u} B[a] \overset{v}{\to} C \right) \overset{+}{\to} \left(S^u \overset{v}{\to} C \right), \tag{2.192}$$

with computation rule: $\mathsf{rec}^v_{S^u}(C,f)\,((a \overset{u}{,} b)) \equiv f(a)(b) : C$.

The induction principle is:

$$\mathsf{ind}^v_{S^u} : \overset{=}{\prod_{C:S^u \overset{\times}{\to} \mathcal{U}_i}} \left(\prod_{a:A}^{v\circ u} \prod_{b:B[a]}^{v} C\,((a \overset{u}{,} b)) \right) \overset{+}{\to} \left(\prod_{e:S^u}^{v} C(e) \right), \tag{2.193}$$

with computation rule: $\mathsf{ind}^v_S(C,f)\,((a \overset{u}{,} b)) \equiv f(a)(b) : C\,((a \overset{u}{,} b))$.

Remark 2.13.6 (Currying)**.** If we have a function $\prod^v_{e:S^u} C(e)$, we can **curry** it to a function $\prod^{v\circ u}_{a:A} \prod^v_{b:B[a]} C\,((a \overset{u}{,} b))$:

$$\mathsf{curry} :\equiv f \overset{+}{\mapsto} \left(a \overset{v\circ u}{\mapsto} b \overset{v}{\mapsto} f\,((a \overset{u}{,} b)) \right) : \left(\prod_{e:S^u}^{v} C(e) \right) \overset{+}{\to} \left(\prod_{a:A}^{v\circ u} \prod_{b:B[a]}^{v} C\,((a \overset{u}{,} b)) \right). \tag{2.194}$$

We can also **uncurry**:

$$\mathsf{uncurry} :\equiv g \overset{+}{\mapsto} (\mathsf{ind}^v_S(C, g)) : \left(\prod_{a:A}^{v\circ u} \prod_{b:B[a]}^{v} C\left((a \overset{u}{,} b)\right)\right) \to \left(\prod_{e:S^u}^{v} C(e)\right). \tag{2.195}$$

So we get

$$\left(\prod_{e:S^u}^{v} C(e)\right) \overset{+}{\simeq} \left(\prod_{a:A}^{v\circ u} \prod_{b:B[a]}^{v} C\left((a \overset{u}{,} b)\right)\right). \tag{2.196}$$

The currying and uncurrying functions are isovariant in the types A and C and the type family $B[a]$.

By variance currying the constructor, it is easy to see that

$$\sum_{x:A}^{-} B[x] \overset{+}{\simeq} \sum_{x:A^{\mathsf{op}}}^{+} B[\mathsf{unflip}\, x], \qquad \sum_{x:A}^{\times} B[x] \overset{+}{\simeq} \sum_{x:A^{\mathsf{core}}}^{+} B[\mathsf{unstrip}\, x]. \tag{2.197}$$

We cannot do a similar thing for $S^=$ because gather has no inverse. Therefore, we will only further investigate S^+ and $S^=$. We will often write (a, b) instead of $(a \overset{+}{,} b)$.

Remark 2.13.7 (Projections from S^+)**.** We define coordinate projections

$$\begin{aligned} \mathsf{prl} &:\equiv \mathsf{rec}^+_{S^+}(A, a \overset{+}{\mapsto} b \overset{+}{\mapsto} a) : S^+ \overset{+}{\to} A, \\ \mathsf{prr} &:\equiv \mathsf{ind}^+_{S^+}(e \overset{\times}{\mapsto} B[\mathsf{prl}\, e], a \overset{+}{\mapsto} b \overset{+}{\mapsto} b) : \prod_{e:S}^{+} B[e]. \end{aligned} \tag{2.198}$$

We have

$$\mathsf{prl}\,((a \overset{+}{,} b)) \equiv a : A, \qquad \mathsf{prr}\,((a \overset{+}{,} b)) \equiv b : B[a]. \tag{2.199}$$

The coordinate projections are isovariant in the type A and the type family $B[a]$. Assuming $e \equiv (\mathsf{prl}\, e \overset{+}{,} \mathsf{prr}\, e)$, they are as strong as the induction principle. Indeed, we have

$$\prod_{e:S^+}^{=} \mathsf{ind}^v_{S^+}(C, f)(e) = f(\mathsf{prl}\, e)(\mathsf{prr}\, e). \tag{2.200}$$

If we want to prove this, the induction principle tells us that we need only prove:

$$\prod_{a:A}^{=} \prod_{b:B[a]}^{=} \mathsf{ind}^v_{S^+}(C, f)\left((a \overset{+}{,} b)\right) = f(\mathsf{prl}\,(a \overset{+}{,} b))(\mathsf{prr}\,(a \overset{+}{,} b)) \tag{2.201}$$

but both left and right hand side are judgementally equal to $f(a)(b)$, so we conclude

this from reflexivity of $=$:

$$a \overset{=}{\mapsto} b \overset{=}{\mapsto} \mathsf{refl}(f(a)(b)) : \prod_{a:A}^{=} \prod_{b:B[a]}^{=} f(a)(b) = f(a)(b). \tag{2.202}$$

Remark 2.13.8 (Projections from $S^=$)**.** We cannot do the same for $S^=$ as we did for S^+ because the recursion principle needs a function that is isovariant in a, and the function $a \overset{+}{\mapsto} b \overset{+}{\mapsto} a$ is not. So instead, we project to the localization of A:

$$\mathsf{prl} :\equiv \mathsf{rec}^+_{S^=}(A^{\mathsf{loc}}, a \overset{=}{\mapsto} b \overset{+}{\mapsto} \mathsf{gather}\, a) : S^= \overset{4}{\to} A^{\mathsf{loc}}. \tag{2.203}$$

It computes: $\mathsf{prl}\,((a \overset{=}{,} b)) \equiv \mathsf{gather}\, a$. The projection prl is isovariant in the type A and the type family $B[a]$.

The right projection cannot be defined, because now we cannot write $B[\mathsf{prl}\, e]$ since $\mathsf{prl}\, e$ is not in A but in A^{loc}. The reason is that $S^=$ may have equated elements from different types $B[a]$, so we can no longer tell to what type $B[a]$ they should be projected. In fact, we will show in section 3.8.7 that if $B[a]$ is covariant, then $S^=$ is the injective limit of $B[a]$ indexed by $a : A$.

Clearly, prl alone is not as strong as the induction principle.

Remark 2.13.9. The reader may wonder why we allow pairs to have various variances in the first component, but only covariance in the second. However, if we need a type of pairs (a, b) that are covariant in a and, e.g., isovariant in b, we can take $\sum^+_{a:A} B(a)^{\mathsf{loc}}$. This is not possible for the first component, as $B(a)$ cannot be written in terms of $\mathsf{gather}\, a$.

2.14 Inductive type families: some examples

In symmetric HoTT

A **type family** is a function $X \to \mathcal{U}_i$ whose function values are types. An **inductive type family** [Uni13, §5.7] is a generalization of an inductive type: we define an entire family of types $P : A \to \mathcal{U}_i$, parametrized over A, at once.

Example 2.14.1. We mentioned before that $P(a)$ can be seen as a proposition about the term a. As an example of an inductive type family, we will define a family $\mathsf{isEven} : \mathbb{N} \to \mathcal{U}_0$ so that we can construct an element of $\mathsf{isEven}(n)$ precisely when n is even.

Inductive type family 2.14.2. We define $\mathsf{isEven} : \mathbb{N} \to \mathcal{U}_0$ with the following constructors:

- $\mathsf{even}_0 : \mathsf{isEven}(\mathsf{zero})$,
- $\mathsf{even}_{+2} : \prod_{n:\mathbb{N}} \mathsf{isEven}(n) \to \mathsf{isEven}(\mathsf{succ}(\mathsf{succ}\, n))$.

The first constructor states that 0 is even, and the other one states that when n is even, then so is $n+2$. The recursion and induction principles are the principles that we might expect if we had not defined $\mathsf{isEven}(n)$ but rather an inductive type T which is isomorphic to $\sum_{n:\mathbb{N}} \mathsf{isEven}(n)$. Such a type would have constructors

- $\mathsf{t}_0 : T$,
- $\mathsf{t}_{+2} : T \to T$,

since creating a term of $\mathsf{isEven}(n)$ for a particular n is a special case of creating a term of T, and for even_{+2}, we can take the two arguments $n : \mathbb{N}$ and $p : \mathsf{isEven}(n)$ of even_{+2} together as a pair $(n, p) : T$.

Note that T has the same definition as the naturals, so it also has the same recursion principle:

$$\mathsf{rec}_T : \prod_{C:\mathcal{U}_i} C \to (T \to C \to C) \to (T \to C). \tag{2.204}$$

To obtain the recursion principle for isEven, we simply replace T with $\sum_{n:\mathbb{N}} \mathsf{isEven}(n)$ (and do a bit of currying) to obtain

$$\mathsf{rec}_{\mathsf{isEven}} : \prod_{C:\mathcal{U}_i} C \to \left(\prod_{n:\mathbb{N}} \mathsf{isEven}(n) \to C \to C\right) \to \left(\prod_{n:\mathbb{N}} \mathsf{isEven}(n) \to C\right). \tag{2.205}$$

Then a function $f :\equiv \mathsf{rec}_{\mathsf{isEven}}(C, c_0, c_{+2})$ is computed as follows:

$$f(0, \mathsf{even}_0) \equiv c_0 : C, \tag{2.206}$$
$$f(n+2, \mathsf{even}_{+2}(n, p)) \equiv c_{+2}(n, p, f(n, p)) : C. \tag{2.207}$$

Here p is a proof that n is even.

The induction principle of T is that of the naturals:

$$\mathsf{ind}_T : \prod_{C:T\to\mathcal{U}_i} C(\mathsf{t}_0) \to \left(\prod_{x:T} C(x) \to C(\mathsf{t}_{+2}\, x)\right) \to \left(\prod_{x:T} C(x)\right). \tag{2.208}$$

Again, we replace T with $\sum_{n:\mathbb{N}} \mathsf{isEven}(n)$. This means C will take an argument $n : \mathbb{N}$ and an argument $p : \mathsf{isEven}(n)$. The constructors of $\mathsf{isEven}(n)$ yield only the latter, so we have to supplement them with the proper natural number:

$$\begin{aligned}\mathsf{ind}_{\mathsf{isEven}} : & \prod_{C:\prod_{n:\mathbb{N}} \mathsf{isEven}(n)\to\mathcal{U}_i} C(0, \mathsf{even}_0) \\ & \to \left(\prod_{n:\mathbb{N}} \prod_{p:\mathsf{isEven}(n)} C(n,p) \to C(n+2, \mathsf{even}_{+2}(n,p))\right) \to \left(\prod_{n:\mathbb{N}} \prod_{p:\mathsf{isEven}(n)} C(n,p)\right).\end{aligned} \tag{2.209}$$

Let's see how we should read this induction principle. In order to construct, for all $n : \mathbb{N}$ at once, a function $f(n) : \prod_{p:\mathsf{isEven}(n)} C(n,p)$, we need to provide

- a value for $f(0)(\mathsf{even}_0) : C(0, \mathsf{even}_0)$,
- whenever p proves that n is even, a value for

$$f(n+2)(\mathsf{even}_{+2}(n,p)) : C(n+2, \mathsf{even}_{+2}(n,p)), \tag{2.210}$$

  but we may assume that $f(n)(p) : C(n,p)$ is already defined.

The induction principle computes just like the recursion principle.

This inspires the following guideline:

Guideline 2.14.3. Instead of defining a single type $T : \mathcal{U}_k$ inductively, we may define a type family $E : \Theta_1 \to \ldots \to \Theta_n \to \mathcal{U}_k$ inductively. In that case:

- Every constructor is a (dependent) function that yields values of type $E(\theta_1, \ldots, \theta_n)$, where each θ_i may depend on the values of the arguments given to the constructor.
- The induction principle, which produces functions of the form

$$f : \prod_{\theta_1:\Theta_1} \cdots \prod_{\theta_n:\Theta_n} \prod_{e:E(\theta_1,\ldots,\theta_n)} C(\theta_1, \ldots, \theta_n, e), \tag{2.211}$$

  takes as arguments a type family C and one function f_i for every constructor χ_i. The function f_i has all arguments from χ_i and, in addition, for every recursive argument $\prod_{e:E(\theta_1,\ldots,\theta_n)}$ (where all θ_j are expressions of type Θ_j) of χ_i, an argument $\prod_{c:C(\theta_1,\ldots,\theta_n,e)}$ used to pass the 'already defined value' $f(e)$ to f_i.
- The computation rules are a straightforward generalization of those for inductive types.

Often, we can also arrive at the correct recursion and induction principles by translating the definition of E into a definition of $T \simeq \sum_{\theta_1:\Theta_1} \ldots \sum_{\theta_n:\Theta_n} E(\theta_1, \ldots, \theta_n)$ and then uncurrying the recursion and induction principles for T.

Example 2.14.4. We will now define types of fixed-length vectors:

Inductive type family 2.14.5. For every type $A : \mathcal{U}_i$, we define the type family $\mathsf{Vec}_\square\, A : \mathbb{N} \to \mathcal{U}_i$ with the following constructors:

- $\ddagger : \mathsf{Vec}_{\mathsf{zero}}\, A$,
- $:: : A \to \prod_{n:\mathbb{N}} \mathsf{Vec}_n\, A \to \mathsf{Vec}_{\mathsf{succ}\, n}\, A$.

The corresponding type $T \simeq \sum_{n:\mathbb{N}} \mathsf{Vec}_n\, A$ is simply the type of lists over A:

- $\ddagger : \mathsf{List}\, A$,
- $:: : A \to \mathsf{List}\, A \to \mathsf{List}\, A$.

Its induction principle was:

$$\mathsf{ind}_{\mathsf{List}\, A} : \prod_{C:\mathsf{List}\, A \to \mathcal{U}_k} C(\ddagger) \to \left(\prod_{a:A} \prod_{\alpha:\mathsf{List}\, A} C(\alpha) \to C(a :: \alpha) \right) \to \left(\prod_{e:\mathsf{List}\, A} C(e) \right), \tag{2.212}$$

which translates to:

$$\mathsf{ind}_{\mathsf{Vec}\, A} : \prod_{C:\prod_{n:\mathbb{N}} \mathsf{Vec}_n\, A \to \mathcal{U}_k} C(\mathsf{zero}, \ddagger) \to \left(\prod_{a:A} \prod_{n:\mathbb{N}} \prod_{\alpha:\mathsf{Vec}_n\, A} C(n, \alpha) \to C(\mathsf{succ}\, n, a :: \alpha) \right) \to \left(\prod_{n:\mathbb{N}} \prod_{e:\mathsf{Vec}_n\, A} C(n, e) \right). \tag{2.213}$$

This is in line with the guideline. It computes:

$$\mathsf{ind}_{\mathsf{Vec}\, A}(C, r, s)(\mathsf{zero}, \ddagger) \equiv r, \tag{2.214}$$

$$\mathsf{ind}_{\mathsf{Vec}\, A}(C, r, s)(\mathsf{succ}\, n, a :: \alpha) \equiv s\left(a, n, \alpha, \mathsf{ind}_{\mathsf{Vec}\, A}(C, r, s)(n, \alpha)\right). \tag{2.215}$$

Note that in $\mathsf{Vec}_n\, A$, the objects A and n play a fundamentally different role here: for every type A, $\mathsf{Vec}_n\, A$ is an inductive type family indexed by $n : \mathbb{N}$. For every type A, we have constructors that create vectors of some length, making use of vectors of different lengths. For every type A, we have a recursion principle that generates functions $\mathsf{Vec}_n\, A \to C$ for *all* n *at once*. We could have defined $\mathsf{Vec}_n\, A$ as an inductive type family indexed by both A and n, but we couldn't have defined all $\mathsf{Vec}_n\, A$'s as separate inductive types unless we took a very different approach, such as taking a single constructor

- $\mathsf{tuple} : (\mathsf{Fin}\, n \to A) \to \mathsf{Vec}_n\, A$.

In directed HoTT

Example 2.14.6. As a first step, we generalize the vector type family to the directed case.

Inductive type family 2.14.7. For every type $A : \mathcal{U}_i$, we define the type family $\mathsf{Vec}_\square\, A : \mathbb{N} \xrightarrow{4} \mathcal{U}_i$, which is covariant in A, with the following constructors:

- $\ddagger : \mathsf{Vec}_{\mathsf{zero}}\, A$,
- $:: : A \xrightarrow{+} \prod^4_{n:\mathbb{N}} \mathsf{Vec}_n\, A \xrightarrow{+} \mathsf{Vec}_{\mathsf{succ}\, n}\, A$.

The corresponding type $T \simeq \sum^4_{n:\mathbb{N}} \mathsf{Vec}_n\, A$ is simply the type of lists over A:

- $\ddagger : \mathsf{List}\, A$,
- $:: : A \xrightarrow{+} \mathsf{List}\, A \xrightarrow{+} \mathsf{List}\, A$.

Its induction principle was:

$$\mathsf{ind}^v_{\mathsf{List}\,A} : \prod_{C:\mathsf{List}\,A \overset{\times}{\to} \mathcal{U}_k}{}^{=} C(\ddagger) \overset{+}{\to} \left(\prod_{a:A}^{+} \prod_{\alpha:\mathsf{List}\,A}^{+} C(\alpha) \overset{+}{\to} C(a :: \alpha) \right) \overset{+}{\to} \left(\prod_{e:\mathsf{List}\,A}^{v} C(e) \right), \tag{2.216}$$

which translates to:

$$\begin{aligned} \mathsf{ind}^v_{\mathsf{Vec}\,A} : \prod_{C:\prod^4_{n:\mathbb{N}} \mathsf{Vec}_n\,A \overset{\times}{\to} \mathcal{U}_k}{}^{=} & C(\mathsf{zero}, \ddagger) \overset{+}{\to} \left(\prod_{a:A}^{+} \prod_{n:\mathbb{N}}^{4} \prod_{\alpha:\mathsf{Vec}_n\,A}^{+} C(n, \alpha) \overset{+}{\to} C(\mathsf{succ}\,n, a :: \alpha) \right) \\ & \overset{+}{\to} \left(\prod_{n:\mathbb{N}}^{4} \prod_{e:\mathsf{Vec}_n\,A}^{v} C(n, e) \right). \end{aligned} \tag{2.217}$$

Unfortunately, we don't get a lot of information from this. Indeed, the only question we need to answer is what is the variance in the index $n : \mathbb{N}$, but $\mathbb{N}$ is a bridgeless groupoid, so we cannot see. To find out, we will define for every type $A : \mathcal{U}_k$ a type family $\mathsf{Fac} : \prod^?_{a,b,c:A} (a \rightsquigarrow_A c) \overset{?}{\to} \mathcal{U}_k$ so that $\mathsf{Fac}(a, b, c, \varphi)$ is the proposition that $\varphi : a \rightsquigarrow c$ factorizes at b, i.e. there are morphisms $\chi : a \rightsquigarrow_A b$ and $\psi : b \rightsquigarrow_A c$ so that $\varphi = \psi \circ \chi$. The Curry-Howard correspondence allows us to simply write this proposition as

$$\mathsf{Fac}(a, b, c, \varphi) :\equiv \sum_{\chi:a\rightsquigarrow b}^{+} \sum_{\psi:b\rightsquigarrow c}^{+} (\varphi =_{a\rightsquigarrow c} \psi \circ \chi). \tag{2.218}$$

From this, we can infer the variance of Fac. Indeed, $\sum^+_{x:X} Y(x)$ is covariant in X and Y; $x \rightsquigarrow y$ is contravariant in x and covariant in y; and $x =_X y$ will be covariant in X and invariant in x and y (that was the reason we needed the concept of invariance in the first place). So we see that a is used contravariantly, b is used invariantly (as it appears both co- and contravariantly), c is used covariantly and φ is used invariantly:

$$\mathsf{Fac} : \prod_{a:A}^{-} \prod_{b:A}^{\times} \prod_{c:C}^{+} (a \rightsquigarrow c) \overset{\times}{\to} \mathcal{U}_k. \tag{2.219}$$

Actually, the variances in a, b and c are not surprising: if we have morphisms $\alpha : a' \rightsquigarrow a$ and $\gamma : c \rightsquigarrow c'$ we do get the implication

$$\mathsf{Fac}(a, b, c, \varphi) \overset{+}{\to} \mathsf{Fac}(a', b, c', \gamma \circ \varphi \circ \alpha). \tag{2.220}$$

A morphism $b \rightsquigarrow b'$ or $b' \rightsquigarrow b$, on the other hand, doesn't give us anything.

But suppose that we know that φ factorizes as $\psi \circ \chi$. Then we also know

$$\mathsf{Fac}(a, b, c, \varphi) = \mathsf{Fac}(a, b, c, \psi \circ \mathsf{id}\,b \circ \chi), \tag{2.221}$$

which we can obtain from $\mathsf{Fac}(b, b, b, \mathsf{id}\,b)$ by exploiting variance. So the only thing we need to know, is that the identity morphism factorizes. This inspires an inductive type definition.

Example 2.14.8.

Inductive type family 2.14.9. For any type $A : \mathcal{U}_i$, we define the type family $\mathsf{Fac} : \prod^{-}_{a:A} \prod^{\times}_{b:A} \prod^{+}_{c:A} (a \rightsquigarrow c) \xrightarrow{\times} \mathcal{U}_i$, which is covariant in A, with the following constructors:

- $\mathsf{fac} : \prod^{=}_{a:A} \mathsf{Fac}(a, a, a, \mathsf{id}\, a)$,
- $\mathsf{Fac}(a, b, c, \varphi)$ is contravariant in a,
- $\mathsf{Fac}(a, b, c, \varphi)$ is covariant in c.

The isovariance of fac is a choice we are allowed to make, but it is also nicely in line with Guideline 2.4.2 as we just want $\mathsf{Fac}(a, a, a, \mathsf{id}\, a)$ to be true and the argument a is just there to assert that the proposition exists.

And yes, that's right, we view contravariance in a and covariance in c as constructors. Neglecting higher order structure, we could write them as follows:

$$\square^* : \prod^{=}_{a',a,b,c:A} \prod^{=}_{\varphi:a\rightsquigarrow c} \prod^{+}_{\alpha:a'\rightsquigarrow a} \mathsf{Fac}(a, b, c, \varphi) \xrightarrow{+} \mathsf{Fac}(a', b, c, \varphi \circ \alpha),$$

$$\square_* : \prod^{=}_{a,b,c,c':A} \prod^{=}_{\varphi:a\rightsquigarrow c} \prod^{+}_{\gamma:c\rightsquigarrow c'} \mathsf{Fac}(a, b, c, \varphi) \xrightarrow{+} \mathsf{Fac}(a, b, c', \gamma \circ \varphi). \tag{2.222}$$

These are the **transport functions** for Fac along the given morphisms: they turn morphisms in a covariant index, into functions. We shall write $\square^*(a', a, b, c, \varphi, \alpha, q)$ as $\alpha^*(q)$. As per Guideline 2.4.2, these functions should be isovariant in a', a, b, c, c' and φ as these objects are there just to assert the existence of other types. In fact, isovariance in all of these arguments will follow from the directed transport lemma 3.3.8 and Lemma 3.2.3.

To create a v-variant function

$$f : \prod^{?}_{a,b,c:A} \prod^{?}_{\varphi:a\rightsquigarrow c} \prod^{v}_{q:\mathsf{Fac}(a,b,c,\varphi)} C(a, b, c, \varphi, q), \tag{2.223}$$

we need:

$$f_{\mathsf{fac}} : \prod^{=}_{a:A} C(a, a, a, \mathsf{id}\, a, \mathsf{fac}\, a),$$

$$f_{\mathsf{left}} : \prod^{=}_{a',a,b,c:A} \prod^{=}_{\varphi:a\rightsquigarrow c} \prod^{v}_{\alpha:a'\rightsquigarrow a} \prod^{v}_{q:\mathsf{Fac}(a,b,c,\varphi)} C(a, b, c, \varphi, q) \xrightarrow{+} C(a', b, c, \varphi \circ \alpha, \alpha^*(q)),$$

$$f_{\mathsf{right}} : \prod^{=}_{a,b,c,c':A} \prod^{=}_{\varphi:a\rightsquigarrow c} \prod^{v}_{\gamma:c\rightsquigarrow c'} \prod^{v}_{q:\mathsf{Fac}(a,b,c,\varphi)} C(a, b, c, \varphi, q) \xrightarrow{+} C(a, b, c', \gamma \circ \varphi, \gamma_*(q)).$$

Now when $v \equiv +$, f_{left} and f_{right} are simply the transport functions $\Box^*$ and $\Box_*$ for C (compare with the ones we had for Fac). So we really need f_{fac} and a type family C that has the proper variance.

In general, f_{left} and f_{right} state that C reverts resp. preserves morphisms v-variantly, i.e. it is $(-_0 v_1)$-variant in a and $(+_0 v_1)$-variant in C. These variances are not always natively available, so we need to replace them with something stronger (as it is a *restriction* we are imposing on C). If $v \in \{\times, +\}$, this just boils down to contra- and covariance. Otherwise, we have to go for isovariance. So the induction principle becomes:

$$\begin{aligned}\mathsf{ind}^v_{\mathsf{Fac}} : & \overset{=}{\prod_{C:\prod_{a:A}^{-\circ w}\prod_{b:A}^{\times}\prod_{c:C}^{w}\prod_{\varphi:a\rightsquigarrow c}^{\times}}} \left(\overset{=}{\prod_{a:A}} C(a,a,a,\mathsf{id}\,a,\mathsf{fac}\,a)\right) \\ & \overset{+}{\to} \left(\overset{?}{\prod_{a,b,c:A}}\ \overset{?}{\prod_{\varphi:a\rightsquigarrow c}}\ \overset{v}{\prod_{q:\mathsf{Fac}(a,b,c,\varphi)}} C(a,b,c,\varphi,q)\right)\end{aligned} \tag{2.224}$$

where $w \equiv +$ if $v \in \{\times, +\}$ and $w \equiv =$ otherwise.

A function f defined as $\mathsf{ind}^v_{\mathsf{Fac}}(C, f_{\mathsf{fac}})$ is computed:

$$\begin{aligned} f(a,a,a,\mathsf{id}\,a,\mathsf{fac}\,a) &\equiv f_{\mathsf{fac}}(a), \\ f(a',b,c,\varphi\circ\alpha,\alpha^*(q)) &\equiv \alpha^*(f(a,b,c,\varphi,q)), \\ f(a,b,c',\gamma\circ\varphi,\gamma_*(q)) &\equiv \gamma_*(f(a,b,c,\varphi,q)). \end{aligned} \tag{2.225}$$

The first line is a real computation rule, the rest is a bit eerie. What the second line actually says is that $f(a',b,c,\Box,\Box)$ and $f(a,b,c,\Box,\Box)$ are, after transporting them to a common type, equal when there is a morphism $\alpha : a' \rightsquigarrow_A a$. In other words: it says that f is isovariant in a. Similarly, the third line says that f is isovariant in c. For the invariant index b, we can think of a 'constructor' that maps $p : b \rightsquigarrow_A b'$ to a relation between $\mathsf{Fac}(a,b,c,\varphi)$ and $\mathsf{Fac}(a,b',c,\varphi)$ and then reason by the same lines, but on less steady ground. An isovariant index would have mapped morphisms to equivalences and is analogous as well. This prompts us to conclude that we should replace the question marks in (2.224) with $=$.

We conclude the section with an extra guideline generalizing Guideline 2.14.3 to the directed case:

Guideline 2.14.10. When we define an inductive type family, we may choose the variance in its indices.

- The covariant and invariant induction principles require destination type families that have the same variance in their indices.
- The contravariant and isovariant induction principles, require a destination type family that is isovariant in all of its indices.[a]

- The functions produced by the induction principle are isovariant in all indices. This is in line with Guideline 2.4.2, as the indices are there just to assert the existence of the destination type.

[a] This is possibly too strict for invariant indices – that would depend on the structure of bridge types, which are problematic: see section 3.5.

2.15 The identity type

In symmetric HoTT

This section is mostly based on [Uni13, §1.12].

The Curry-Howard correspondence between propositions and types, needs a type $a =_A b$ corresponding to the proposition that $a : A$ equals $b : A$. The elements of this type, are the proofs that $a = b$. There are many ways to prove that two things are equal. We have the reflexivity, symmetry and transitivity laws that turn equality into an equivalence relation; we know that when $a = b$, then $f(a) = f(b)$ and when $f = g$, then $f(a) = g(a)$ etc. It turns out that we only have to state reflexivity as a constructor; the induction principle will prove the rest.

Inductive type 2.15.1. When $A : \mathcal{U}_i$ is a type, we define the family of **identity types** $\square =_A \square : A \to A \to \mathcal{U}_i$ with the following constructor:

- $\mathsf{refl} : \prod_{a:A} a =_A a$.

Elements of $a =_A b$ are called **paths** from a to b. When there is a path from a to b, we say that a and b are **propositionally equal** (as opposed to judgementally equal: $a \equiv b$).

As in the previous section, the recursion and induction principles will look a lot like the principles for a type $T(A)$ isomorphic to $\sum_{a,b:A} a =_A b$ (which is an abbreviation for $\sum_{a:A} \sum_{b:A} a =_A b$). This type would have the following constructor:

- $\mathsf{t} : A \to T(A)$.

Note that this means that a term of $T(A)$ is simply a term of A in a box. The recursion and induction principles are entirely boring:

$$\mathsf{rec}_{T(A)} : \prod_{C:\mathcal{U}_k} (A \to C) \to (T(A) \to C), \tag{2.226}$$

$$\mathsf{ind}_{T(A)} : \prod_{C:T(A)\to\mathcal{U}_k} \left(\prod_{a:A} C(\mathsf{t}\, a) \right) \to \left(\prod_{x:T(A)} C(x) \right). \tag{2.227}$$

The computation rules are

$$\begin{aligned} &\mathsf{rec}_{T(A)}(C, f)(\mathsf{t}\, a) \equiv f(a) : C, \\ &\mathsf{ind}_{T(A)}(C, f)(\mathsf{t}\, a) \equiv f(a) : C(\mathsf{t}\, a). \end{aligned} \tag{2.228}$$

In order to obtain the recursion principle for the identity types, we replace $T(A)$ with $\sum_{a,b:A} a =_A b$ and do some uncurrying:

$$\mathsf{rec}_{=_A} : \prod_{C:\mathcal{U}_k} (A \to C) \to \left(\prod_{a,b:A} (a =_A b) \to C \right). \tag{2.229}$$

To obtain the induction principle, we do the same. Then C becomes a dependent type in three arguments and we replace $C(\mathsf{t}\, a)$ with $C(a, a, \mathsf{refl}\, a)$:

$$\mathsf{ind}_{=_A} : \prod_{C:\prod_{a,b:A}(a=_A b)\to\mathcal{U}_k} \left(\prod_{a:A} C(a, a, \mathsf{refl}\, a) \right) \to \left(\prod_{a,b:A} \prod_{p:a=_A b} C(a, b, p) \right). \tag{2.230}$$

We can read this principle as follows: in order to prove that $C(a, b, p)$ holds for all $a, b : A$ and $p : a =_A b$, we may assume that a and b are the same term and p is simply reflexivity. This idea is called **path induction**, as is the object $\mathsf{ind}_{=_A}$. The computation rules are

$$\begin{aligned} &\mathsf{rec}_{=_A}(C, f)(a, a, \mathsf{refl}\, a) \equiv f(a) : C, \\ &\mathsf{ind}_{=_A}(C, f)(a, a, \mathsf{refl}\, a) \equiv f(a) : C(a, a, \mathsf{refl}\, a). \end{aligned} \tag{2.231}$$

Just like $A \times B$, A^{op}, A^{core} and A^{loc}, the type family $=_A$ has only one constructor. Thus, we can try to find an analogue of currying and projecting. The analogue of currying yields

$$\left(\prod_{a:A} C(a, a, \mathsf{refl}\, a) \right) \simeq \left(\prod_{a,b:A} \prod_{p:a=_A b} C(a, b, p) \right), \tag{2.232}$$

where the induction principle takes you to the right, and evaluation in $a, a, \mathsf{refl}\, a$ takes you back to the left. The projections were always created by feeding the identity function(s) to the induction principle. We can try that here:

$$\pi :\equiv \mathsf{ind}_{=_A}(A, \mathrm{id}_A). \tag{2.233}$$

It computes: $\pi(a, a, \mathsf{refl}\, a) \equiv a$. But what is $\pi(a, b, p)$ when p is not reflexivity? We can prove that $\pi(a, b, p) = a$, and we can prove that $\pi(a, b, p) = b$. This is not a contradiction, as p proves that $a = b$. In topological terms $\pi(a, b, p)$ as a mysterious point on the path p that connects a and b.

Based path induction Above, we defined the identity types as an inductive type family in two arguments. However, we might as well have taken one end of the paths fixed:

Inductive type family 2.15.2. When $A : \mathcal{U}_i$ is a type and $a_0 : A$, we the **identity type family based at** a_0, denoted $!a_0 =_A \square : A \to \mathcal{U}_i$, as the inductive type family with the following constructor:

- $\mathsf{brefl}_{a_0} : !a_0 =_A a_0$.

Note that brefl_{a_0} takes no arguments at all.

This definition yields the **based path induction** principle

$$\mathsf{ind}_{!a_0=_A} : \prod_{C:\prod_{b:A}(!a_0=_A b)\to\mathcal{U}_k} C(a_0, \mathsf{brefl}\, a_0) \to \left(\prod_{b:A}\prod_{p:!a_0=_A b} C(b,p)\right). \tag{2.234}$$

This says that, when we have a type $C(b,p)$ for every $b : A$ with $p : !a_0 =_A b$, then in order to prove that $C(b,p)$ always holds, it suffices to prove that $C(a_0, \mathsf{brefl}\, a_0)$ holds. It computes:

$$\mathsf{ind}_{!a_0=_A}(C,c)(a_0, \mathsf{brefl}\, a_0) \equiv c : C(a_0, \mathsf{brefl}\, a_0). \tag{2.235}$$

In fact, it does not matter: path induction and based path induction are equally strong, so that we can use whichever suits best. This is expressed by the following lemma:

Lemma 2.15.3. For all $a, b : A$, the types $(a =_A b)$ and $(!a =_A b)$ are equivalent:

$$\prod_{a,b:A} (a =_A b) \simeq (!a =_A b). \tag{2.236}$$

We will give the proof in the directed case a bit further down.

Applying path induction As an example of what these identity types are capable of, we prove that applying functions preserves propositional equality [Uni13, §2.2]:

Lemma 2.15.4. Suppose $f : A \to B$ and $a, a' : A$ are propositionally equal. Then $f(a)$ and $f(a')$ are propositionally equal.

In mathematics founded on set theory, when we state a theorem, we mean to say "This proposition is provable" and below follows a sketch of how the theorem is proven using the inference rules of propositional logic. In type theory, we mean to say "The type corresponding to this proposition, has a term", and below we construct that term. Since there is no judgement asserting that a type has a term without explicitly giving it, the proof is an integral part of the assertion. In this proof, we will be defining a function of the type

$$\prod_{a,a':A}\prod_{p:a=_A a'} (f(a) =_B f(a')). \tag{2.237}$$

Note that $(f(a) =_B f(a'))$ does not depend on p, so we didn't need a *dependent* function type there, but this way it is easier to recognize the opportunity for using path induction.

Proof. By path induction, it is sufficient to construct a function of type $\prod_{a:A}(f(a) =_B f(a))$. We take $a \mapsto \mathsf{refl}\, f(a)$.

More formally, for every $x, y : A$ and $p : x =_A y$, we define $C(x,y,p) :\equiv (f(x) =_B f(y)) : \mathcal{U}_i$. This defines the type family C. We have $g :\equiv a \mapsto \mathsf{refl}\, f(a) : \prod_{a:A} C(a,a,\mathsf{refl}\, a)$. This yields

$$\mathsf{ind}_{=}(C,g) : \prod_{a,a':A}\prod_{p:a=_A a'} C(a,a',p) \tag{2.238}$$

which is what we wanted to find. □

In directed HoTT

Inductive type 2.15.5. When $A : \mathcal{U}_i$ is a type (covariantly), we define the family of **identity types** $\square =_A \square : A \overset{\times}{\to} A \overset{\times}{\to} \mathcal{U}_i$, which is covariant in A, with the following constructor:

- $\mathsf{refl} : \prod^{=}_{a:A} a =_A a$.

The invariance of $a =_A b$ in a and b is a choice, but it was the very reason for introducing invariant functions, so it seems a good choice. The covariance of the identity type family in A, follows from the fact that A only occurs covariantly in the constructor's argument types. The fact that we choose reflexivity to be isovariant is in line with Guideline 2.4.2, as we just want $a =_A a$ to be true and the argument a is only there to assert the existence of said type. The analogous case of the morphism type will provide a more convincing argument.

The induction principle is:

$$\mathsf{ind}^{v}_{=_A} : \prod_{C:\prod^{\times}_{a,b:A}(a=_A b)\overset{\times}{\to}\mathcal{U}_k}^{=} \left(\prod_{a:A}^{=} C(a,a,\mathsf{refl}\,a)\right) \overset{+}{\to} \left(\prod_{a,b:A}^{=} \prod_{p:a=_A b}^{v} C(a,b,p)\right). \tag{2.239}$$

Here, isovariance of the 'induction basis' $\prod^{=}_{a:A} C(a,a,\mathsf{refl}\,a)$ follows from the fact that refl is isovariant and $v \circ = \equiv =$. The produced function is isovariant in a and b by Guideline 2.14.10 and v-variant in p because we are using the v-variant induction principle.

The induction principle computes:

$$\mathsf{ind}^{v}_{=_A}(C,f)(a,a,\mathsf{refl}\,a) \equiv f(a) : C(a,a,\mathsf{refl}\,a). \tag{2.240}$$

Lemma 2.15.6. Every identity type is a bridgeless groupoid:

$$\prod_{A:\mathcal{U}_k}^{=} \prod_{a,b:A}^{=} \mathsf{isBridgelessGrpd}(A). \tag{2.241}$$

Proof. By definition of a bridgeless groupoid, we need to prove

$$\prod_{A:\mathcal{U}_k}^{=} \prod_{a,b:A}^{=} \sum_{f:a=b\overset{=}{\to}a=b}^{+} (f_+ = \mathrm{id}_{a=b}). \tag{2.242}$$

We first construct the isovariant identity function

$$g :\equiv \mathsf{ind}^{=}_{=_A}(a \overset{\times}{\mapsto} b \overset{\times}{\mapsto} p \overset{\times}{\mapsto} a =_A b, \mathsf{refl}) : \prod_{a,b:A}^{=} (a=b) \overset{=}{\to} (a=b). \tag{2.243}$$

Then $g(a,b) : (a=b) \overset{=}{\to} (a=b)$, and it computes: $g(a,a,\mathsf{refl}\,a) \equiv \mathsf{refl}\,a : a =_A a$.

It remains to show that $\prod^{=}_{a,b:A}(g(a,b)_+ = \mathrm{id}_{a=b})$. By the function extensionality axiom (section 3.8.8), we need only prove

$$\prod_{a,b:A}^{=} \prod_{p:a=b}^{=} (g(a,b)_+(p) = p). \tag{2.244}$$

Then by path induction, we just need $\prod^{=}_{a:A}(g(a,a)_+(\mathsf{refl}\, a) = \mathsf{refl}\, a)$. But if we compute the left hand side:

$$g(a,a)_+(\mathsf{refl}\, a) \equiv \left(p \overset{+}{\mapsto} g(a,a,p)\right)(\mathsf{refl}\, a) \equiv g(a,a,\mathsf{refl}\, a) \equiv \mathsf{refl}\, a, \tag{2.245}$$

we find that we just need reflexivity: $a \overset{=}{\to} \mathsf{refl}(\mathsf{refl}\, a)$. □

Based path induction Again, we can define a based identity type family:

Inductive type family 2.15.7. When $A : \mathcal{U}_i$ is a type and $a_0 : A$, we the **identity type family based at** a_0, which is covariant in A and invariant in a_0, denoted $!a_0 =_A \square : A \overset{\times}{\to} \mathcal{U}_i$, as the inductive type family with the following constructor:

- $\mathsf{brefl}_{a_0} : !a_0 =_A a_0$.

As constructors are always isovariant in the objects used to build the type (e.g. $\mathsf{inl} : A \overset{+}{\to} A + B$ is isovariant in A), brefl_{a_0} is isovariant in a_0. What is remarkable is that $!a_0 =_A \square$ is *not* isovariant in a_0 and A. Guideline 2.7.4 on page 32 restricts the variance in a_0 and A based on how they appear in the constructors' argument types, but the constructor doesn't take any arguments, so we would expect $!a_0 =_A \square$ to be isovariant in a_0 and A. The reason that it is invariant in a_0 has to do with the fact that a_0 appears in the index (the right hand side) of the output type of brefl_{a_0}. The covariance in A has to do with the fact that the index a_0 has type A. We come back to this in the next section on morphism types.

We get the based path induction principle:

$$\mathsf{ind}^{v}_{!a_0=_A} : \prod_{C:\prod^{\times}_{b:A}(!a_0=_A b)\overset{\times}{\to}\mathcal{U}_k}^{=} C(a_0, \mathsf{brefl}\, a_0) \overset{+}{\to} \left(\prod_{b:A}^{=} \prod_{p:!a_0=_A b}^{v} C(b,p)\right). \tag{2.246}$$

Again, this induction principle is as good as the non-based one:

Lemma 2.15.8. For all $a, b : A$, the types $(a =_A b)$ and $(!a =_A b)$ are equivalent:

$$\prod_{a,b:A}^{=} (a =_A b) \overset{4}{\simeq} (!a =_A b). \tag{2.247}$$

Proof. We can write $\overset{4}{\simeq}$ because $a =_A b$ is a bridgeless groupoid.

We first create a function

$$f : \prod_{a,b:A}^{=} (a =_A b) \overset{4}{\to} (!a =_A b) \tag{2.248}$$

to the right. By path induction, we just need

$$\prod_{a:A}^{=} !a =_A a, \tag{2.249}$$

and here we can take $a \overset{=}{\mapsto} \mathsf{brefl}_a$, as brefl_a is isovariant in a. Then f computes: $f(a, a, \mathsf{refl}\, a) \equiv \mathsf{brefl}_a : !a =_A a$.

Now we construct

$$g : \prod_{a,b:A}^{=} (!a =_A b) \overset{4}{\to} (a =_A b). \tag{2.250}$$

By based path induction, we need, for every $a :^{=} A$, an element of $a =_A a$, and there we can take $\mathsf{refl}\, a$ which is isovariant in a. Then $g(a)$ computes: $g(a)(a, \mathsf{brefl}_a) \equiv \mathsf{refl}\, a : a =_A a$.

We have to show that they are inverses. First, we show

$$\prod_{a,b:A}^{=} \prod_{p:a=_A b}^{=} (g(a,b) \circ f(a,b))(p) =_{a=_A b} p. \tag{2.251}$$

By path induction, we just need

$$\prod_{a:A}^{=} (g(a,a) \circ f(a,a))(\mathsf{refl}\, a) = \mathsf{refl}\, a. \tag{2.252}$$

But the left hand side computes:

$$(g \circ f)(a, a, \mathsf{refl}\, a) \equiv g(a, a, f(a, a, \mathsf{refl}\, a)) \equiv g(a, a, \mathsf{brefl}_a) \equiv \mathsf{refl}\, a, \tag{2.253}$$

so we just need $\mathsf{refl}(\mathsf{refl}\, a)$, which is indeed isovariant in a.

From the other side, we need to show

$$\prod_{a,b:A}^{=} \prod_{p:!a=_A b}^{=} (f(a,b) \circ g(a,b))(p) =_{!a=_A b} p. \tag{2.254}$$

By based path induction, we need, for every $a :^{=} A$, an element of

$$(f(a,a) \circ g(a,a))(\mathsf{brefl}_a) = \mathsf{brefl}_a. \tag{2.255}$$

But the left hand side computes:

$$(f(a,a) \circ g(a,a))(\mathsf{brefl}_a) \equiv f(a, a, g(a)(a, \mathsf{brefl}_a)) \equiv f(a, a, \mathsf{refl}\, a) \equiv \mathsf{brefl}_a, \tag{2.256}$$

so we just need $\mathsf{refl}(\mathsf{brefl}_a)$, which is indeed isovariant in a. □

2.16 The morphism type

The morphism type family only exists in the directed case. Its definition differs only from the one of the identity type family in its indices' variance:

Definition 2.16.1. When $A :^+ \mathcal{U}_i$ is a type, we define the family of **morphism types** $\square =_A \square : A \xrightarrow{-} A \xrightarrow{+} \mathcal{U}_i$, which is covariant in A, as the inductive type family with the following constructors:

- $\mathsf{id} : \prod^{=}_{a:A} a \rightsquigarrow_A a$,
- the type family $a =_A b$ is contravariant in a,
- the type family $a =_A b$ is covariant in b.

The covariance of the identity type family in A, follows from the fact that A only occurs covariantly in the constructor's argument types. The fact that we choose the identity morphism $\mathsf{id}\, a$ to be isovariant in a is in line with Guideline 2.4.2, as we just want $a \rightsquigarrow_A a$ to be true and the argument a is only there to assert the existence of said type. However, there is a more convincing argument. Suppose that we have a morphism $\varphi : a \rightsquigarrow_A b$. We get $\mathsf{id}\, a : a \rightsquigarrow_A a$ and $\mathsf{id}\, b : b \rightsquigarrow_A b$. If we want to compare these morphisms, we have to transport them along φ to a middle ground $a \rightsquigarrow_A b$:

$$\varphi_*(\mathsf{id}\, a) = \varphi \circ \mathsf{id}\, a, \qquad \varphi^*(\mathsf{id}\, b) = \mathsf{id}\, b \circ \varphi. \tag{2.257}$$

Now we do expect these morphisms to be equal. Then id should be an isovariant function.

The morphism induction principle differs only from the path induction principle in the restrictions it imposes on the variance of the destination type family:

$$\mathsf{ind}^v_{\rightsquigarrow_A} : \prod_{C:\prod^{-\circ w}_{a:A}\prod^{w}_{b:B}(a\rightsquigarrow_A b)\xrightarrow{\times}\mathcal{U}_k}^{=} \left(\prod_{a:A}^{=} C(a,a,\mathsf{id}\, a)\right) \xrightarrow{+} \left(\prod_{a,b:A}^{=}\prod_{\varphi:a\rightsquigarrow_A b}^{v} C(a,b,\varphi)\right), \tag{2.258}$$

where $w \equiv +$ if $v \in \{\times, +\}$, and $w \equiv =$ if $v \in \{-, =\}$. It computes:

$$\mathsf{ind}^v_{\rightsquigarrow_A}(C, f)(a, a, \mathsf{id}\, a) \equiv f(a) : C(a, a, \mathsf{id}\, a). \tag{2.259}$$

Based morphism induction We can define morphism types based at the source and based at the target:

Inductive type family 2.16.2. Given a type $A : \mathcal{U}_i$ and an element $a_0 : A$, we define the **morphism type family based at source** a_0, denoted $!a_0 \rightsquigarrow_A \square : A \xrightarrow{+} \mathcal{U}_i$, which is covariant in A and contravariant in a_0, with the following constructors:

- $\mathsf{sbid}_{a_0} : !a_0 \rightsquigarrow_A a_0$,
- $!a_0 \rightsquigarrow b$ is covariant in b.

Inductive type family 2.16.3. Given a type $A : \mathcal{U}_i$ and an element $b_0 : A$, we define the **morphism type family based at target** b_0, denoted $\square \rightsquigarrow_A !b_0 : A \xrightarrow{-} \mathcal{U}_i$, which is covariant in A and in b_0, with the following constructors:

- $\mathsf{tbid}_{b_0} : b_0 \rightsquigarrow_A !b_0$,
- $a \rightsquigarrow !b_0$ is contravariant in a.

As with the based identity type, it is surprising that the based morphism types are not isovariant in A and their base, even though their constructors don't take any arguments. Let's start with looking at the base.

When we want to construct a function $f' : A + B \xrightarrow{+} A' + B$ from $f : A \xrightarrow{+} A'$, we just map $\mathsf{inl}\, a$ to $\mathsf{inl}\, f(a)$ and $\mathsf{inr}\, b$ to itself. The analogous thing for the source-based morphism type would be the following: given a morphism $\varphi : a_0 \rightsquigarrow a_0'$, we map the sole constructor sbid_{a_0} to $\mathsf{sbid}_{a_0'}$ (note that the index a_0 is analogous to the hidden indices A and B on $\mathsf{inl}_{A,B}$). Indeed, these would give us an equivalence between $!a_0 \rightsquigarrow \square$ and $!a_0' \rightsquigarrow \square$ as we could also map $\mathsf{sbid}_{a_0'}$ back to sbid_{a_0}.

There is a problem here, however, and that is that this alleged equivalence doesn't respect the indices: we need, for every $b : A$, an equivalence $(!a_0 \rightsquigarrow b) \overset{+}{\simeq} (!a_0' \rightsquigarrow b)$. So the image of $\mathsf{sbid}_{a_0'} : !a_0' \rightsquigarrow a_0'$ should be in $!a_0 \rightsquigarrow a_0'$, so that $\mathsf{sbid}_{a_0} : !a_0 \rightsquigarrow a_0$ is not acceptable as an image.

Of course we do have a transport function $\varphi_* : (!a_0 \rightsquigarrow a_0) \xrightarrow{+} (!a_0 \rightsquigarrow a_0')$ so that we can map $\mathsf{sbid}_{a_0'} : !a_0' \rightsquigarrow a_0'$ to $\varphi_*(\mathsf{sbid}_{a_0}) : (!a_0 \rightsquigarrow a_0')$, but not the other way around, as we do not have $(!a_0' \rightsquigarrow a_0') \xrightarrow{+} (!a_0' \rightsquigarrow a_0)$. So we see that because a_0 is used *covariantly* in the index of the output type of the constructor, the type family based at a_0 is *contravariant* in a_0. For the target-based morphism type, we would get a dual conclusion. We capture this in a guideline:

Guideline 2.16.4. When a type family T depends v-variantly on an object x (which is *not* an index of the type family), then x can only be used $(- \circ v)$-variantly in the indices of the output types of the constructors.

Then the question remains why these type families are covariant, rather than isovariant, in A. Here, we need to consider the transport 'constructor' again:

$$\Box_* : \prod_{b,b':A}^{=} \prod_{\beta:b\rightsquigarrow_A b'}^{+} (!a_0 \rightsquigarrow_A b) \xrightarrow{+} (!a_0 \rightsquigarrow_A b'). \tag{2.260}$$

This constructor takes an argument of type $b \rightsquigarrow_A b'$, which depends covariantly on A. Then the based type's variance in A must not be stronger than $+$. Indeed, if it were isovariant in A, then a function $f : A \xrightarrow{+} C$ would yield an equivalence $(!a_0 \rightsquigarrow_A b) \overset{+}{\simeq} (!f(a_0) \rightsquigarrow_C f(b))$ but we would have problems going from right to left: it is precisely the image of a transport along $\chi : f(b) \rightsquigarrow f(b')$ that is problematic. Even in the identity type, we have a kind of hidden transport constructor

$$\Box_* : \prod_{b,b':A}^{=} \prod_{p:b=_A b'}^{+} (!a_0 =_A b) \overset{+}{\simeq} (!a_0 =_A b'), \tag{2.261}$$

which enforced covariance in A as $b =_A b'$ is only covariant in A. We can conclude:

Guideline 2.16.5. When a type family T depends on an object x which appears covariantly in one of the indices' types, it must be either covariant or invariant in x.

The induction principles are:

$$\mathsf{ind}^v_{!a_0\rightsquigarrow_A} : \prod_{C:\prod^w_{b:A}(!a_0\rightsquigarrow_A b)\xrightarrow{\times}\mathcal{U}_k}^{=} C(a_0, \mathsf{sbid}_{a_0}) \xrightarrow{+} \left(\prod_{b:A}^{=} \prod_{\varphi:!a_0\rightsquigarrow_A b}^{v} C(b,\varphi) \right), \tag{2.262}$$

$$\mathsf{ind}^v_{\rightsquigarrow_A!b_0} : \prod_{C:\prod^{-\circ w}_{a:A}(a\rightsquigarrow_A !b_0)\xrightarrow{\times}\mathcal{U}_k}^{=} C(b_0, \mathsf{tbid}_{b_0}) \xrightarrow{+} \left(\prod_{a:A}^{=} \prod_{\varphi:a\rightsquigarrow_A !b_0}^{v} C(a,\varphi) \right), \tag{2.263}$$

where $w \equiv +$ if $v \in \{+, \times\}$ and $w \equiv =$ if $v \in \{=, -\}$. They compute:

$$\mathsf{ind}^v_{!a_0\rightsquigarrow_A}(C,c)(a_0, \mathsf{sbid}_{a_0}) \equiv c : C(a_0, \mathsf{sbid}_{a_0}), \tag{2.264}$$

$$\mathsf{ind}^v_{\rightsquigarrow_A!b_0}(C,c)(b_0, \mathsf{tbid}_{b_0}) \equiv c : C(b_0, \mathsf{tbid}_{b_0}). \tag{2.265}$$

We prove an analogue of lemma 2.15.8:

Lemma 2.16.6. For all $a, b : A$, the types $(a \rightsquigarrow_A b)$, $(!a \rightsquigarrow_A b)$ and $(a \rightsquigarrow_A !b)$ are equivalent:

$$\prod_{a,b:A}^{=} (a \rightsquigarrow_A b) \overset{+}{\simeq} (!a \rightsquigarrow_A b), \qquad \prod_{a,b:A}^{=} (a \rightsquigarrow_A b) \overset{+}{\simeq} (a \rightsquigarrow_A b!). \tag{2.266}$$

Proof. We only prove the first equivalence. We first create a function

$$f : \prod_{a,b:A}^{=} (a \rightsquigarrow_A b) \xrightarrow{+} (!a \rightsquigarrow_A b) \tag{2.267}$$

to the right. The type $!a \rightsquigarrow_A b$ is contravariant in a and covariant in b, so by morphism induction, we just need

$$\prod_{a:A}^{=} !a \rightsquigarrow_A a, \tag{2.268}$$

and here we can take $a \overset{=}{\mapsto} \mathsf{sbid}_a$, as sbid_a is isovariant in a. Then f computes: $f(a, a, \mathsf{id}, a) \equiv \mathsf{sbid}_a : !a \rightsquigarrow_A a$.

Now we construct

$$g : \prod_{a,b:A}^{=} (!a \rightsquigarrow_A b) \xrightarrow{+} (a \rightsquigarrow_A b). \tag{2.269}$$

The type $(a \rightsquigarrow_A b)$ is covariant in b, so by source-based morphism induction, we need, for every $a :^{=} A$, an element of $a \rightsquigarrow_A a$, and there we can take $\mathsf{id}\, a$ which is isovariant in a. Then $g(a)$ computes: $g(a)(a, \mathsf{sbid}_a) \equiv \mathsf{id}\, a : a \rightsquigarrow_A a$.

We have to show that they are inverses. First, we show

$$\prod_{a,b:A}^{=} \prod_{\varphi:a\rightsquigarrow_A b}^{4} (g(a,b) \circ f(a,b))(\varphi) =_{a=_A b} \varphi. \tag{2.270}$$

The destination type is covariant in $a \rightsquigarrow_A b$ and therefore contravariant in a and covariant b. By morphism induction, we just need

$$\prod_{a:A}^{=} (g(a,a) \circ f(a,a))(\mathsf{id}\, a) = \mathsf{id}\, a. \tag{2.271}$$

The left hand side computes:

$$(g \circ f)(a, a, \mathsf{id}\, a) \equiv g(a, a, f(a, a, \mathsf{id}\, a)) \equiv g(a, a, \mathsf{sbid}_a) \equiv \mathsf{id}\, a, \tag{2.272}$$

so we just need $\mathsf{refl}(\mathsf{id}\, a)$, which is indeed isovariant in a.

From the other side, we need to show

$$\prod_{a,b:A}^{=} \prod_{\varphi:!a\rightsquigarrow_A b}^{4} (f(a,b) \circ g(a,b))(\varphi) =_{!a\rightsquigarrow_A b} \varphi. \tag{2.273}$$

Again, the destination type is indirectly covariant in b, so by source-based morphism induction, we need, for every $a :^{=} A$, an element of

$$(f(a,a) \circ g(a,a))(\mathsf{sbid}_a) = \mathsf{sbid}_a. \tag{2.274}$$

But the left hand side computes:

$$(f(a,a) \circ g(a,a))(\mathsf{sbid}_a) \equiv f(a,a,g(a)(a,\mathsf{sbid}_a)) \equiv f(a,a,\mathrm{id}\,a) \equiv \mathsf{sbid}_a, \qquad (2.275)$$

so we just need $\mathsf{refl}(\mathsf{sbid}_a)$, which is indeed isovariant in a. □

2.17 W-types

In symmetric HoTT

This section is based on [Uni13, §5.3].

As mentioned in section 2.7, our general way of defining inductive types, does not combine well with the discussion about judgements in section 2.1. More precisely, there are no inference rules that allow us to define a new inductive type. However, one can show that, using a few basic inductive types that are included out-of-the-box, one can construct types equivalent to most inductive types.

We already demonstrated to some extent that the disjoint union $\sum_{a:A} P(a)$ of an inductive type family $\lambda a.P(a)$ is isomorphic to an inductive type. Identity types allow us to go back to the type family. For example, one can define a function $\mathsf{length} : \mathsf{List}\,A \to \mathbb{N}$ and show that

$$\mathsf{Vec}_n\,A \simeq \sum_{e:\mathsf{List}\,A} \mathsf{length}\,e =_{\mathbb{N}} n. \qquad (2.276)$$

When an inductive type T has several constructors, all of them without recursive arguments, then we can write T as a coproduct of inductive types with only one constructor. Similarly, when S has one constructor with multiple non-recursive arguments, we can show that S is isomorphic to the type of dependent tuples of these arguments' types.

The only remaining problem is that we have no way to construct inductive types which have a recursive constructor. This is resolved when we have W-types or well-founded tree types.

Inductive type 2.17.1. When A is a type and $B[x]$ is a type depending on a variable $x : A$, we define the type $\mathrm{W}_{x:A} B[x]$ of **well-founded trees** as the inductive type with the following constructor:

- $\mathsf{sup} : \prod_{a:A} (B[a] \to \mathrm{W}_{x:A} B[x]) \to \mathrm{W}_{x:A} B[x]$.

The constructor sup is called the **supremum** function.

So to construct a term of $\mathrm{W}_{x:A} B[x]$, we have to provide a term $a : A$ and then $B[a]$-many recursive arguments.

When we have an inductive type T and we want to show it is isomorphic to a W-type, we will identify a term t with $\mathsf{sup}(a, f)$, where a encodes what constructor t was created with, as well as all non-recursive arguments given to that constructor. The type $B[a]$ encodes the number of recursive arguments required by this constructor, and f gives them.

For example, if we want to prove that $\mathbb{N}$ is isomorphic to some $\mathrm{W}_{x:A} B[x]$, we can take $A = \mathsf{Fin}(2)$, where 0_2 stands for the constructor 0, and 1_2 stands for succ. The constructor 0 takes no recursive arguments, so $B[0_2] = \mathbf{0}$. The other one takes one, so $B[1_2] = \mathbf{1}$.

We now have all basic types needed (and more) to construct almost any inductive type. However, while defining inductive types explicitly in terms of these basic types makes things more precise, it also makes them less readable, so we will continue to work with definitions of inductive types. As such, we will not be using W-types and we do not consider them in the directed case.

2.18 Higher inductive types

In symmetric HoTT

This section is based on [Uni13, ch.6].

Higher inductive types are a generalization of inductive types. When we define a higher inductive type A, we can give constructors that construct terms of type A, but we may also construct paths between terms of A, or paths between such paths, or... Mostly as an illustration, we present a few higher inductive types in this section.

In topology, we have the topological space $[0,1]$, which has the property that a path in any space A is the same a function $p : [0,1] \to A$. More specifically, we regard p as a path from $p(0)$ to $p(1)$. We have a correspondence between spaces and types and in fact, higher inductive types allow us to define an analog:

Definition 2.18.1. The **interval type** I is defined as the higher inductive type with constructors [Uni13, §6.3]:

- $0_I : I$,
- $1_I : I$,
- $\mathsf{seg}_I : 0_I =_I 1_I$.

In order to define a function $f : I \to C$, we need to provide values $f(0_I)$ and $f(1_I)$ and prove that they are equal. The recursion principle looks like this:

$$\mathsf{rec}_I : \prod_{C:\mathcal{U}_i} \prod_{c_0,c_1:C} c_0 =_C c_1 \to (I \to C)\,, \tag{2.277}$$

with computation rules

$$\mathsf{rec}_I(C, c_0, c_1, p)(0_I) \equiv c_0 : C, \tag{2.278}$$
$$\mathsf{rec}_I(C, c_0, c_1, p)(1_I) \equiv c_1 : C, \tag{2.279}$$
$$\mathsf{rec}_I(C, c_0, c_1, p)^{=}(\mathsf{seg}_I) \equiv p : c_0 = c_1. \tag{2.280}$$

The induction principle is similar, but the provided path will not connect $c_0 : C(0_I)$ to $c_1 : C(1_I)$, but rather the transport $q_*(c_0) : C(1_I)$ to $c_1 : C(1_I)$, where $q :\equiv C^{=}(\mathsf{seg}_I)$ proves that $C(0_I) = C(1_I)$. One can prove that

$$(I \to C) \simeq \left(\sum_{c_0,c_1:C} c_0 = c_1 \right). \tag{2.281}$$

The left-to-right function is given by evaluating $f : I \to C$ in $0_I, 1_I$ and seg_I. An inverse is constructed from the recursion principle.

Higher inductive types allow us to do "synthetic" algebraic topology: instead of constructing a topological space as a certain set of points with a certain structure, we simply state a few principles that describe the space. For example, we describe the circle S^1 as a space with a point base and a non-trivial path loop from base to base.

Definition 2.18.2. The **circle** S^1 is defined as the higher inductive type with constructors [Uni13, §6.4]:

- $\mathsf{base} : S^1$,
- $\mathsf{loop} : \mathsf{base} = \mathsf{base}$.

The sphere S^2 can be described as a space with a point base and a non-trivial homotopy surf from $\mathsf{refl}\,\mathsf{base}$ to $\mathsf{refl}\,\mathsf{base}$:

Definition 2.18.3. The **sphere** S^2 is defined as the higher inductive type with constructors [Uni13, §6.4]:

- $\mathsf{base} : S^2$,
- $\mathsf{surf} : \mathsf{refl}\,\mathsf{base} = \mathsf{refl}\,\mathsf{base}$.

A third example of how higher inductive types are useful, is the following:

Definition 2.18.4. We define the **propositional truncation** $\|A\|$ of a type A as the higher inductive type with constructors:

- $|\square| : A \to \|A\|$,
- $\mathsf{trunc} : \prod_{a,b:\|A\|} a = b$.

So if we have a term $a : A$, we get a term $|a| : \|A\|$, but we make no distinction between terms of $\|A\|$. If P is a proposition, then $\|P\|$ is read as "Merely P". The constructor $|\square|$ proves that P implies $\|P\|$, but $\|P\|$ has at most one proof (up to propositional equality). So when we work with truncated propositions, we are no longer doing constructive, proof relevant mathematics, but rather mathematics in a more classical logic.

In directed HoTT

> Although in practice, higher inductive types with morphism constructors will be a substantial ingredient to obtain types with an interesting category structure, we leave them to future work.

Chapter 3

Homotopy type theory, symmetric and directed

At this point, the theories in which we will be doing symmetric and directed HoTT, are defined. We will be extending them by an occasional inductive definition, but the content of chapter 2 will be the core of the theories. In this chapter, we find out what we can do with them. Mostly, we will be porting basic facts about HoTT to the directed case, closely following [Uni13, ch.2].

In general, there will be two opportunities for doing this. The first one is that in directed HoTT, we have a directed analogue $a \rightsquigarrow_A b$ of the identity type $a =_A b$, which equips any type with a higher category structure, instead of the ∞-groupoid structure of types in symmetric HoTT. So we will get directed versions of theorems about this ∞-groupoid structure. For example, in symmetric HoTT, we know that every function preserves equalities. In directed HoTT, we will be able to prove that every covariant function preserves equalities covariantly.

Secondly, there is one type in symmetric HoTT that already has a higher category structure: the universe, with functions as its morphisms. So for some theorems about the universe in symmetric HoTT, we will get analogous theorems in directed HoTT that hold in *any* type. For example, in symmetric HoTT, we can prove that when two types are equal, $A =_{\mathcal{U}_k} B$, then they are equivalent, $A \simeq B$, meaning that there is an invertible function between them. Similarly, in directed HoTT, we can prove that when two objects are equal, $a =_A b$, then they are isomorphic, $a \cong_A b$, meaning that there is an invertible morphism between them.

Most results from symmetric HoTT simply hold in directed HoTT. We will state them with variance in the symmetric HoTT section. The directed HoTT section will be for non-straightforward generalizations and completely new results.

In section 3.7, we give a formal treatment of groupoids in directed HoTT. Until then, we will not be using the variance annotations 3 and 4.

3.1 Types are higher categories

This section is based on [Uni13, §2.1].

In symmetric HoTT

We have mentioned before that types are ∞-groupoids. In fact, they are weak ∞-groupoids, where the word 'weak' means that they obey the groupoid laws only up to isomorphism, or, in type theoretic language, up to propositional equality. We can also view them as homotopical spaces. The correspondence is as follows:

HoTT	Higher category theory	Homotopy theory
type A	∞-groupoid $\mathcal{A}$	homotopical space A
element $x : A$	object $x \in \mathsf{obj}(\mathcal{A})$	point $x \in A$
type $a =_A b$	∞-groupoid $\mathrm{Hom}(a, b)$	path space from a to b
$p : a =_A b$	isomorphism $p : a \to b$	path p from a to b
$h : p =_{a=b} q$	isomorphism $h : p \to q$	homotopy h from p to q.

The intuitive content of this correspondence is expressed the following lemma, which also holds in directed HoTT. We state and prove it in directed HoTT, the version for symmetric HoTT is obtained by removing all variance annotations.

Lemma 3.1.1. For any type $A :^{=} \mathcal{U}_k$, the following hold:

1. Propositional equality is reflexive / there is an identity isomorphism in every object / there is a constant path at any point:

$$\mathsf{refl} : \prod_{a:A}^{=} a =_A a. \tag{3.1}$$

2. Propositional equality is symmetric / isomorphisms/paths are invertible:

$$\square^{-1} : \prod_{a,b:A}^{=} (a =_A b) \xrightarrow{+} (b =_A a). \tag{3.2}$$

3. Propositional equality is transitive / isomorphisms/paths can be composed:

$$\circ : \prod_{a,b,c:A}^{=} (b =_A c) \xrightarrow{+} (a =_A b) \xrightarrow{+} (a =_A c). \tag{3.3}$$

These are laws on the lowest level: the assert that objects are equal. On the next level, we get laws that assert that isomorphisms/paths are equal:

4. Reflexivity is the unit for composition, up to isomorphism/homotopy:

$$\prod_{a,b}^{=} \prod_{p:a=b}^{+} \mathsf{refl}\, b \circ p = p, \qquad \prod_{a,b}^{=} \prod_{p:a=b}^{+} p \circ \mathsf{refl}\, a = p. \tag{3.4}$$

5. An isomorphism composed with its inverse is isomorphic to refl / a path composed with its inverse is homotopic to refl:

$$\prod_{a,b}^{=}\prod_{p:a=b}^{+} p^{-1}\circ p = \mathsf{refl}\, a, \qquad \prod_{a,b}^{=}\prod_{p:a=b}^{+} p\circ p^{-1} = \mathsf{refl}\, b. \tag{3.5}$$

6. Inverting twice does nothing, up to isomorphism/homotopy:

$$\prod_{a,b}^{=}\prod_{p:a=b}^{+} (p^{-1})^{-1} = p. \tag{3.6}$$

7. Composition is associative, up to isomorphism/homotopy:

$$\prod_{a,b,c,d}^{=}\prod_{p:a=b}^{+}\prod_{q:b=c}^{+}\prod_{r:c=d}^{+} r\circ(q\circ p) = (r\circ q)\circ p. \tag{3.7}$$

On the next level, we get laws that assert homotopies between homotopies, or isomorphisms between isomorphisms, e.g.

8. For any chain of equalities

$$a \overset{p}{=\!=} b \overset{q}{=\!=} c \overset{r}{=\!=} d \overset{s}{=\!=} e, \tag{3.8}$$

the composition of the proofs of the following instances of associativity

$$\begin{aligned}&((s\circ r)\circ q)\circ p = (s\circ r)\circ(q\circ p) = s\circ(r\circ(q\circ p))\\ &= s\circ((r\circ q)\circ p) = (s\circ(r\circ q))\circ p = ((s\circ r)\circ q)\circ p\end{aligned} \tag{3.9}$$

is reflexivity.

9. ...

There are infinitely many weak ∞-groupoid laws (the ones at higher levels are called **coherence laws**), but fortunately in HoTT, we can prove them as we need them using path induction.

Proof. 1. This is just the constructor of the identity type.

2. By path induction, we need only prove $\prod^{=}_{a:A} a =_A a$, which is proven by refl. More formally, $\square^{-1} :\equiv \mathsf{ind}^{+}_{=}(a \overset{\times}{\mapsto} b \overset{\times}{\mapsto} p \overset{\times}{\mapsto} b =_A a, \mathsf{refl})$. It computes: $(\mathsf{refl}\, a)^{-1} \equiv \mathsf{refl}\, a$.

3. After swapping arguments, we can apply path induction on the path from b to c so that we only need to find $\prod^{=}_{a,d:A}(a =_A d) \overset{+}{\to} (a =_A d)$ and here, we can take the identity function $\mathrm{id}_{a=d}$. Then composition computes: $(\mathsf{refl}\, b)\circ p \equiv p$.

4. To show that refl is the left inverse, we simply compute: $\mathsf{refl}\, b \circ p \equiv p$. Then we can just take $a \overset{=}{\mapsto} b \overset{=}{\mapsto} p \overset{=}{\mapsto} \mathsf{refl}\, p$. On the right, by path induction, we need only prove

$\prod^{=}_{a:A}(\mathsf{refl}\, a) \circ (\mathsf{refl}\, a) = \mathsf{refl}\, a$. But the left hand side computes to the right hand side, so this is proven by $\mathsf{refl}(\mathsf{refl}\, a)$.

5. Apply path induction and observe that the left hand side computes to the right hand side.
6. Idem.
7. Apply path induction on r and observe that both sides compute to $q \circ p$.
8. Path induction, path induction, path induction. □

Remark 3.1.2. Observe that our definition of the composition function was asymmetric: we applied path induction on the left morphism, resulting in an asymmetric computation rule. The consequence is that the left and right unit laws are proved differently. We could also have defined composition by applying path induction twice. Then it would compute $(\mathsf{refl}\, a) \circ (\mathsf{refl}\, a) \equiv \mathsf{refl}\, a$. One can show that this is the same function, up to propositional equality.

In directed HoTT

By lemma 3.1.1, the identity types still equip every type with a weak ∞-groupoid structure, even in the directed case. In addition to that, the morphism types equip every type with a weak higher category structure. We have the following correspondence with between directed HoTT, higher category theory and directed homotopy theory:

Directed HoTT	Higher category theory	Directed homotopy theory
type A	higher category $\mathcal{A}$	homotopical space A
element $x : A$	object $x \in \mathsf{obj}(\mathcal{A})$	point $x \in A$
type $a \rightsquigarrow_A b$	higher category $\mathrm{Hom}(a, b)$	directed path space from a to b
$\varphi : a \rightsquigarrow_A b$	morphism $\varphi : a \to b$	directed path φ from a to b
type $a =_A b$	subcategory of isomorphisms	path space from a to b
$p : a =_A b$	isomorphism $p : a \to b$	path p from a to b.

This is partly expressed by the following lemmas:

Lemma 3.1.3. For any type $A :^{=} \mathcal{U}_k$, the following hold:

1. There is an identity morphism in every object:
$$\mathsf{id} : \prod_{a:A}^{=} a \rightsquigarrow_A a. \tag{3.10}$$
2. Morphisms can be composed:
$$\circ : \prod_{a,b,c:A}^{=} (b \rightsquigarrow_A c) \xrightarrow{+} (a \rightsquigarrow_A b) \xrightarrow{+} (a \rightsquigarrow_A c). \tag{3.11}$$

These are laws on the lowest level: they assert the existence of morphisms between objects. On the next level, we get laws that assert that morphisms are equal:

3. The identity morphism is the unit for composition:

$$\prod_{a,b}^{=} \prod_{\varphi:a\rightsquigarrow b}^{+} \mathsf{id}\, b \circ \varphi =_{a\rightsquigarrow b} \varphi, \qquad \prod_{a,b}^{=} \prod_{\varphi:a\rightsquigarrow b}^{+} \varphi \circ \mathsf{id}\, a =_{a\rightsquigarrow b} \varphi. \tag{3.12}$$

4. Composition is associative:

$$\prod_{a,b,c,d}^{=} \prod_{\varphi:a\rightsquigarrow b}^{+} \prod_{\chi:b\rightsquigarrow c}^{+} \prod_{\psi:c\rightsquigarrow d}^{+} \psi \circ (\chi \circ \varphi) =_{a\rightsquigarrow d} (\psi \circ \chi) \circ \varphi. \tag{3.13}$$

On the next level, we get laws that assert that equalities between morphisms are equal:

5. For any chain of morphisms

$$a \overset{\varphi}{\rightsquigarrow} b \overset{\chi}{\rightsquigarrow} c \overset{\psi}{\rightsquigarrow} d \overset{\omega}{\rightsquigarrow} e, \tag{3.14}$$

the composition of the proofs of the following instances of associativity

$$\begin{aligned} &((\omega \circ \psi) \circ \chi) \circ \varphi = (\omega \circ \psi) \circ (\chi \circ \varphi) = \omega \circ (\psi \circ (\chi \circ \varphi)) \\ &= \omega \circ ((\psi \circ \chi) \circ \varphi) = (\omega \circ (\psi \circ \chi)) \circ \varphi = ((\omega \circ \psi) \circ \chi) \circ \varphi \end{aligned} \tag{3.15}$$

is reflexivity.

6. ...

There are infinitely many weak higher category laws, but fortunately in directed HoTT, we can prove them as we need them using morphism induction.

Proof. 1. This is just the constructor of the morphism type.

2. Since $\prod^{=}_{a:A}(a \rightsquigarrow_A b) \overset{+}{\to} (a \rightsquigarrow_A c)$ is contravariant in b and covariant in c, we can apply covariant morphism induction on the morphism $b \rightsquigarrow_A c$. Then we need to prove $\prod^{=}_{a,d:A}(a \rightsquigarrow_A d) \overset{+}{\to} (a \rightsquigarrow_A d)$. Here we can take the identity function $\mathrm{id}_{a\rightsquigarrow d}$. Then composition computes: $(\mathsf{id}\, b) \circ \varphi \equiv \varphi$.

3. By the above computation rule, we only need reflexivity to prove the left unit law. For the right unit law, we can apply covariant morphism induction, because $\varphi \circ \mathsf{id}\, a =_{a\rightsquigarrow b} \varphi$ is contravariant in a and covariant in b. (Note that a and b also appear hiddenly in the composition operator, but only isovariantly). Then we have to prove that $(\mathsf{id}\, a) \circ (\mathsf{id}\, a) = \mathsf{id}\, a$, which is proven by reflexivity.

4. The type $\psi \circ (\chi \circ \varphi) =_{a\rightsquigarrow d} (\psi \circ \chi) \circ \varphi$ is isovariant in c, which only appears as index of the composition operator, and covariant in d via the index of the

identity type. So we can apply covariant morphism induction on ψ, after which both sides compute to $\chi \circ \varphi$.

5. This is proven by repeated morphism induction. □

Moreover, the ∞-groupoid structure induced by the identity type and the higher category structure induced by the morphism type, are related by the following lemma:

Lemma 3.1.4. In any type $A :^{=} \mathcal{U}_k$, a path implies a morphism:

$$\mathsf{toMorph} : \prod_{a,b:A}^{=} (a =_A b) \xrightarrow{+} (a \rightsquigarrow_A b). \tag{3.16}$$

It computes: $\mathsf{toMorph}(\mathsf{refl}\, a) \equiv \mathsf{id}\, a$.

Proof. By covariant path induction, we need only prove $\prod_{a:A}^{=} a \rightsquigarrow_A a$, which is proven by the identity morphism. More formally,

$$\mathsf{toMorph} :\equiv \mathsf{ind}^{+}_{=}(a \overset{\times}{\mapsto} b \overset{\times}{\mapsto} \varphi \overset{\times}{\mapsto} a \rightsquigarrow_A b, \mathsf{id}).$$ □

We can also prove, using path induction, that $\mathsf{toMorph}(a \circ b) = \mathsf{toMorph}(a) \circ \mathsf{toMorph}(b)$, and that $\mathsf{toMorph}$ is well-behaved with respect to the higher coherence laws of higher categories.

In order to understand the above correspondence completely, we actually needed more than the previous lemma: it is not enough that a path implies a morphism; we need paths to be the same things as isomorphisms. We will complete the correspondence in section 3.9

3.2 Functions are functors

In symmetric HoTT

If the type $a =_A b$ is to encode the proposition that a and b are in some sense equal, then we had better prove that for any function $f : A \to C$, we have $(a =_A b) \to (f(a) =_C f(b))$. Translated to categorical language, this means that f preserves isomorphisms, and since isomorphisms are the only morphisms in a groupoid, this means that f is a functor. Translated to homotopy theoretical language, it means that f preserves paths, i.e. it is path continuous[1]. Again, we formulate the directed version of the result. For the undirected version, remove variance annotations.

Lemma 3.2.1. For every two types $A, C :^{=} \mathcal{U}_k$ and every function $f :^{=} A \xrightarrow{v} C$, there is a function

$$f^{=} : \prod_{a,b:A}^{=} (a =_A b) \xrightarrow{+} (f(a) =_C f(b)). \tag{3.17}$$

[1]This seems to be a non-standard term in topology. We call a function between topological spaces **path continuous** if it maps paths to paths path-continuously. As such, every continuous function is path continuous.

We will mostly omit the first two arguments. It computes: $f^{=}(\mathsf{refl}\, a) \equiv \mathsf{refl}\, f(a)$.

Proof. By path induction, we need only prove $\prod^{=}_{a:A} f(a) =_C f(a)$, which is proven by $\mathsf{refl}\, f(a)$. More formally, we define

$$f^{=} :\equiv \mathsf{ind}^{+}_{=}(a \overset{\times}{\mapsto} b \overset{\times}{\mapsto} p \overset{\times}{\mapsto} (f(a) =_C f(b)), a \overset{=}{\mapsto} \mathsf{refl}\, f(a)) \tag{3.18}$$

As the induction principle is isovariant in the destination type family, and $\mathsf{refl}\, f(a)$ is isovariant in $f(a)$, the entire expression happens to be isovariant in f. The computation rule follows immediately. □

Lemma 3.2.2. For every two types $A, C :^{=} \mathcal{U}_k$ and every function $f :^{=} A \overset{v}{\to} C$, the function $f^{=}$ satisfies (among others) the following properties:

1. $\prod^{=}_{x:A} f^{=}(\mathsf{refl}\, x) =_{f(x)=f(x)} \mathsf{refl}\, f(x)$,
2. $\prod^{=}_{x,y,z:A} \prod^{+}_{p:x=y} \prod^{+}_{q:y=z} f^{=}(q \circ p) =_{f(x)=f(z)} f^{=}(q) \circ f^{=}(p)$,
3. $\prod^{=}_{x,y:A} \prod^{+}_{p:x=y} f^{=}(p^{-1}) =_{f(y)=f(x)} f^{=}(p)^{-1}$.

For every type $A :^{=} \mathcal{U}_k$, we have:

4. $\prod^{=}_{x,y:A} \prod^{+}_{p:x=y} \mathrm{id}^{=}_{A}(p) =_{x=y} p$.

For every three types $A, B, C :^{=} \mathcal{U}_k$ and all functions $f :^{=} A \overset{u}{\to} B$ and $g :^{=} B \overset{v}{\to} C$, we have:

5. $\prod^{=}_{x,y:A} \prod^{+}_{p:x=y} (g \circ f)^{=}(p) = (g^{=} \circ f^{=})(p)$.

Proof.

1. As $f^{=}(\mathsf{refl}\, x) \equiv \mathsf{refl}\, f(x)$, this is just reflexivity.
2. By path induction on q, we need only prove $\prod^{=}_{x,y:A} \prod^{+}_{p:x=y} f^{=}(\mathsf{refl}\, y \circ p) =_{f(x)=f(y)} f^{=}(\mathsf{refl}\, y) \circ f^{=}(p)$. On the left, we get $f^{=}(\mathsf{refl}\, y \circ p) \equiv f^{=}(p)$. On the right, we have $f^{=}(\mathsf{refl}\, y) \circ f^{=}(p) \equiv (\mathsf{refl}\, f(y)) \circ f^{=}(p) \equiv f^{=}(p)$, so we need only reflexivity.
3. By path induction, we need only prove $\prod^{=}_{x:A} f^{=}((\mathsf{refl}\, x)^{-1}) = f^{=}(\mathsf{refl}\, x)^{-1}$, but both sides compute to $\mathsf{refl}\, f(x)$.
4. By path induction, we need only prove $\prod^{=}_{x:A} \mathrm{id}^{=}_{A}(\mathsf{refl}\, x) = \mathsf{refl}\, x$, but the left hand side computes to $\mathsf{refl}\, \mathrm{id}_A(x) \equiv \mathsf{refl}\, x$, so this is reflexivity.
5. By path induction, we need only prove $\prod^{=}_{x:A} (g \circ f)^{=}(\mathsf{refl}\, x) = (g^{=} \circ f^{=})(\mathsf{refl}\, x)$. Both sides compute to $\mathsf{refl}\, (g \circ f)(x)$.

As refl is isovariant in both argument and type, and covariant path induction is isovariant in everything but the path, we see that all assumptions can be taken isovariant. □

In directed HoTT

Lemmas 3.2.1 and 3.2.2 also hold in the directed case. Also, functions of different variance have various actions on morphisms.

Lemma 3.2.3. Let $A, C :^{=} \mathcal{U}_k$ be types.

1. For every **covariant** $f :^{=} A \xrightarrow{+} C$, there is a function
$$f^{\rightsquigarrow} : \prod_{x,y:A}^{=} (x \rightsquigarrow_A y) \xrightarrow{+} (f(x) \rightsquigarrow_C f(y)). \tag{3.19}$$
It computes: $f^{\rightsquigarrow}(\mathsf{id}\, x) \equiv \mathsf{id}\, f(x)$.

2. For every **contravariant** $f :^{=} A \xrightarrow{-} C$, there is a function
$$f^{\rightsquigarrow} : \prod_{x,y:A}^{=} (x \rightsquigarrow_A y) \xrightarrow{+} (f(y) \rightsquigarrow_C f(p)). \tag{3.20}$$
It computes: $f^{\rightsquigarrow}(\mathsf{id}\, x) \equiv \mathsf{id}\, f(x)$.

3. For every **isovariant** $f :^{=} A \xrightarrow{=} C$, there is a function
$$f^{\rightsquigarrow} : \prod_{x,y:A}^{=} (x \rightsquigarrow_A y) \xrightarrow{+} (f(x) =_C f(y)). \tag{3.21}$$
It computes: $f^{\rightsquigarrow}(\mathsf{id}\, x) \equiv \mathsf{refl}\, f(x)$.

By lack of a bridge type (see section 3.5), we cannot state an analogue for invariant functions.

Proof. 1. As f is covariant, the type family $f(x) \rightsquigarrow_A f(y)$ is contravariant in x and covariant in y. Then by morphism induction, we need only prove $\prod_{x:A}^{-} f(x) \rightsquigarrow_C f(x)$, which is proven by the identity morphism.

2. As f is contravariant, the type family $f(y) \rightsquigarrow_A f(x)$ is covariant in y and contravariant in x. So we can apply morphism induction again.

3. Although the identity type is invariant, the function f is isovariant so that $f(x) =_A f(y)$ is isovariant in x and y. Then we can apply morphism induction and have to prove $\prod_{x:A}^{=} f(x) =_C f(x)$, which is proven by reflexivity. □

There is an arsenal of theorems that we might want to state about the functorial actions on paths and morphisms. We prove two properties as an example:

Lemma 3.2.4. Let $A, C :^{=} \mathcal{U}_k$ be types, and $f :^{=} A \xrightarrow{-} C$ a contravariant function. Then

1. the functorial action of f respects composition:

$$\prod_{x,y,z:A}^{=} \prod_{\varphi:x\rightsquigarrow y}^{+} \prod_{\chi:y\rightsquigarrow z}^{+} f^{\rightsquigarrow}(\chi \circ \varphi) =_{f(z)\rightsquigarrow f(x)} f^{\rightsquigarrow}(\varphi) \circ f^{\rightsquigarrow}(\chi). \tag{3.22}$$

2. the weakening of paths to morphisms is well-behaved with respect to f:

$$\prod_{x,y:A}^{=} \prod_{p:x=y}^{+} \mathsf{toMorph}\left(f^{=}(p)^{-1}\right) =_{f(y)\rightsquigarrow f(x)} f^{\rightsquigarrow}(\mathsf{toMorph}\, p). \tag{3.23}$$

Proof. 1. We want to apply morphism induction on χ. The composition operators are isovariant in x, y and z, so that is all right. The identity type is covariant in its index, which in this case is $f(z) \rightsquigarrow f(x)$. Since f is contravariant, this index is covariant in z and does not depend on y. So we can apply morphism induction and have to prove

$$\prod_{x,y:A}^{=} \prod_{\varphi:x\rightsquigarrow y}^{+} f^{\rightsquigarrow}(\mathsf{id}\, y \circ \varphi) =_{f(y)\rightsquigarrow f(x)} f^{\rightsquigarrow}(\varphi) \circ f^{\rightsquigarrow}(\mathsf{id}\, y). \tag{3.24}$$

This computes to

$$\prod_{x,y:A}^{=} \prod_{\varphi:x\rightsquigarrow y}^{+} f^{\rightsquigarrow}(\varphi) =_{f(y)\rightsquigarrow f(x)} f^{\rightsquigarrow}(\varphi) \circ \mathsf{id}\, f(y). \tag{3.25}$$

By the same reasoning, we can apply morphism induction again and have to prove

$$\prod_{x:A}^{=} f^{\rightsquigarrow}(\mathsf{id}\, x) =_{f(x)\rightsquigarrow f(x)} f^{\rightsquigarrow}(\mathsf{id}\, x) \circ \mathsf{id}\, f(x), \tag{3.26}$$

where both sides compute to $\mathsf{id}\, f(x)$, so that it is proven by $\mathsf{refl}(\mathsf{id}\, f(x))$.

2. Again, the identity type is covariant in its index and $f(y) \rightsquigarrow f(x)$ is covariant in y and contravariant in x, so we can apply morphism induction and have to prove

$$\prod_{x:A}^{=} \mathsf{toMorph}\left(f^{=}(\mathsf{refl}\, x)^{-1}\right) =_{f(x)\rightsquigarrow f(x)} f^{\rightsquigarrow}(\mathsf{toMorph}(\mathsf{refl}\, x)). \tag{3.27}$$

On the left, we get

$$\begin{aligned} \mathsf{toMorph}\left(f^{=}(\mathsf{refl}\, x)^{-1}\right) &\equiv \mathsf{toMorph}\left(\mathsf{refl}\, f(x)\right)^{-1} \\ &\equiv \mathsf{toMorph}\left(\mathsf{refl}\, f(x)\right) \equiv \mathsf{id}\, f(x) : x \rightsquigarrow_A x. \end{aligned} \tag{3.28}$$

On the right, we get

$$f^{\rightsquigarrow}(\mathsf{toMorph}(\mathsf{refl}\, x)) \equiv f^{\rightsquigarrow}(\mathsf{id}\, x) \equiv \mathsf{id}\, f(x) : x \rightsquigarrow_A x, \tag{3.29}$$

so we just need reflexivity. □

3.3 Equivalences, isomorphisms and transport

This section is based on [Uni13, §2.3 and 2.4].

In symmetric HoTT

In section 2.6, we gave a 'logical definition' for the type $\mathsf{isEquiv}(f)$ which states that $f : A \overset{+}{\to} B$ is invertible:

$$\mathsf{isEquiv}(f)(f) \overset{+}{\leftrightarrows} \sum_{g:B\overset{+}{\to}A}^{+} (g \circ f = \mathrm{id}_A) \times (f \circ g = \mathrm{id}_B). \tag{3.30}$$

We will call the right hand side $\mathsf{qInv}(f)$, the type of **quasi-inverses** of f.

The reason that this is only a logical equivalence and not an equivalence of types, is that a quasi-inverse contains too much information: for a single invertible function $f : A \overset{+}{\to} B$, there may be multiple quasi-inverses [Uni13, §2.4], whereas we would expect a function to have at most a single inverse. We will first give the actual definition of an equivalence, then show that the logical equivalence given above, really holds, and conclude with an informal argument why a quasi-inverse contains too much information.

Definition 3.3.1. Let $A, B :^{\times} \mathcal{U}_k$ be types and $f :^{\times} A \overset{+}{\to} B$ a function. We define the types of **left and right inverses** of f as follows:

$$\mathsf{leftInv}(f) :\equiv \sum_{\ell:B\overset{+}{\to}A}^{+} (\ell \circ f) = \mathrm{id}_A \qquad \mathsf{rightInv}(f) :\equiv \sum_{r:B\overset{+}{\to}A}^{+} (f \circ r) = \mathrm{id}_B. \tag{3.31}$$

We call f an **equivalence** if it has a left and a right inverse:

$$\mathsf{isEquiv}(f) :\equiv \mathsf{leftInv}(f) \times \mathsf{rightInv}(f). \tag{3.32}$$

The type of **equivalences** from A to B is defined:

$$A \overset{+}{\simeq} B :\equiv \sum_{f:A\overset{+}{\to}B}^{+} \mathsf{isEquiv}(f). \tag{3.33}$$

Although we may sometimes reason with equivalences that have a variance different from $+$, we do not define them formally and will not use them in proofs. We use Σ^+-types because they are the natural generalization of Σ-types in symmetric HoTT.

Lemma 3.3.2. Given types $A, B :^{\times} \mathcal{U}_k$, then a function $f :^{\times} A \overset{+}{\to} B$ is an equivalence if and only if it has a quasi-inverse:

$$\mathsf{isEquiv}(f) \overset{+}{\leftrightarrows} \mathsf{qInv}(f). \tag{3.34}$$

Proof. We construct a function in each direction.

($\overset{+}{\leftarrow}$) Take $x :^{+} \mathsf{qInv}(f)$. The induction principles for the Σ-type and the product, allow us to assume that x is of the form $(g, (p, q))$, with $g :^{+} B \overset{+}{\to} A$, $p :^{+} g \circ f = \mathrm{id}_A$ and $q :^{+} f \circ g = \mathrm{id}_B$. Then we map this value to $((g, p), (g, q)) : \mathsf{isEquiv}(f)$.

($\overset{+}{\to}$) Take $x :^{+} \mathsf{isEquiv}(f)$. By induction, we may assume that x is of the form $((\ell, p), (r, q))$. We show that $\ell = r$:

$$\ell = \ell \circ \mathrm{id}_A = \ell \circ (f \circ r) = (\ell \circ f) \circ r = \mathrm{id}_B \circ r = r. \tag{3.35}$$

What is meant by this chain of equalities, is that we can easily construct a proof of each individual equality, and then compose all this proofs to obtain $t : \ell = r$.

By path induction, we can construct a function $\prod^{=}_{A,B:\mathcal{U}_k}(A = B) \overset{+}{\to} (A \overset{+}{\to} B)$, along the lines of the proof of the transport lemma below, but without referring to equivalences. Then, since $\ell = r$, we also know that $((f \circ r) = \mathrm{id}_B) = ((f \circ \ell) = \mathrm{id}_B)$, and so we get a function

$$t_* : ((f \circ r) = \mathrm{id}_B) \overset{+}{\to} ((f \circ \ell) = \mathrm{id}_B). \tag{3.36}$$

So the triple $(\ell, (p, t_*(q)))$ proves $\mathsf{qInv}(f)$. □

The above proof indicates why $\mathsf{qInv}(f)$ contains too much information. One can prove that

$$\mathsf{qInv}(f) \overset{+}{\simeq} \sum_{\lambda:\mathsf{leftInv}(f)}^{+} \sum_{\rho:\mathsf{rightInv}(f)}^{+} \mathsf{prl}\,\lambda =_{B\to A} \mathsf{prl}\,\rho, \tag{3.37}$$

i.e. a quasi-inverse of f consists of a function ℓ with a proof that ℓ is left inverse to f; a function r with a proof that r is right inverse to f; and a proof $s : \ell = r$ – al available covariantly. To go to the right, we map $(g, (p, q))$ to $((g, p), (g, q), \mathsf{refl}\, g)$. To go to the left, we apply path induction on s.

Now in the proof of lemma 3.3.2, we saw that we could already construct a path $t : \ell = r$ without using s, so actually we have two paths from ℓ to r, and that is a problem. Indeed, if we have two objects and a path between them, we can apply path induction and just assume that the objects are really the same, reducing the path to reflexivity. In topological terms: if we have two endpoints and a path between them, we can contract the whole thing to a single point, reducing the path to a constant one. But here, we have two objects with two paths between them. The induction principle allows us to assume that the objects are the same and to reduce one path to reflexivity, but the other one remains. Topologically, when we have two points on a loop, we can move them to the same point, but we cannot necessarily contract the loop. So an element of $\mathsf{qInv}(f)$ proves that f is invertible and in addition gives you an arbitrary and unneeded loop from the inverse to itself.

The type $\mathsf{isEquiv}(f)$, on the contrary, is a mere proposition:

Definition 3.3.3. A type P is a **mere proposition** if any two terms of P are equal [Uni13, §3.3]:

$$\mathsf{isProp}(P) :\equiv \prod_{x,y:P}^{+} x = y. \tag{3.38}$$

The definition is chosen covariant but as the identity types are bridgeless groupoids, this does not matter.

Theorem 3.3.4. For any function f, $\mathsf{isEquiv}(f)$ is a mere proposition.

Proof. For a proof in the symmetric case, see [Uni13, §4.3]. As the proof is extremely involved, we leave this as a conjecture in the directed case. □

Finally, we prove the utterly important transport lemma:

Lemma 3.3.5 (Transport)**.** Equal types are equivalent:

$$\mathsf{transport} : \prod_{A,B:\mathcal{U}_k}^{=} (A =_{\mathcal{U}_k} B) \xrightarrow{+} (A \overset{+}{\simeq} B). \tag{3.39}$$

Proof. By path induction, we need only prove $\prod^{=}_{A:\mathcal{U}_k} A \overset{+}{\simeq} A$. This is proven by the identity equivalence $(\mathrm{id}_A, \mathrm{id}_A, \mathsf{refl}\,\mathrm{id}_A, \mathsf{refl}\,\mathrm{id}_A) : A \overset{+}{\simeq} A$, which has the correct variance as id_A is isovariant in A. □

Notation 3.3.6. If $e : A \overset{+}{\simeq} B$, we denote the corresponding function $A \xrightarrow{+} B$ also as e, and the inverse function $B \xrightarrow{+} A$ as e^{-1}.

When $T : A \xrightarrow{v} \mathcal{U}_k$ is a type family and $p : a =_A b$ is a path in A, then we denote $\mathsf{transport}(T^{=}(p))$ as $p_* : T(a) \overset{+}{\simeq} T(b)$. If T is isovariant and $\varphi : a \rightsquigarrow_A b$ is a morphism, we also denote $\mathsf{transport}(T^{\rightsquigarrow}(\varphi))$ as $\varphi_* : T(a) \overset{+}{\simeq} T(b)$.

Remark 3.3.7. One can show that $\mathsf{transport}$ preserves groupoid structure. In section 3.8.9, the univalence axiom will turn $\mathsf{transport}$ into an equivalence: $(A =_{\mathcal{U}_k} B) \overset{+}{\simeq} (A \overset{+}{\simeq} B)$. Then we can say that an path between types is the same as an equivalence between types.

In directed HoTT

As mentioned in the introduction of this chapter, there are two heuristics for generalizing results from symmetric to directed HoTT. One is to see morphisms as a directed generalization of equality; the other is to see morphisms as a generalization of functions to non-universe types. So far, we have mostly used the former heuristic. In this section, we will be able to use both. We start with the former. The only references to the identity type in the symmetric part of this section, were in the definition of equality and in the transport lemma. If we want to replace the

equalities with morphisms in the definition of an equivalence, we are thinking about adjoint functions, which we do not treat here. The transport lemma has a directed generalization:

Lemma 3.3.8 (Directed transport)**.** A morphism between types, implies a covariant function:

$$\mathsf{dirTransport} : \prod_{A,B:\mathcal{U}_k}^{=} (A \rightsquigarrow_{\mathcal{U}_k} B) \xrightarrow{+} (A \xrightarrow{+} B). \tag{3.40}$$

Proof. By morphism induction, we need only prove $\prod^{=}_{A:\mathcal{U}_k} A \xrightarrow{+} A$. This is proven by the identity function id_A, which is indeed isovariant in A. □

Notation 3.3.9. When $T : A \xrightarrow{+} \mathcal{U}_k$ is a covariant type family and $\varphi : a \rightsquigarrow_A b$ is a morphism in A, we write $\varphi_* :\equiv \mathsf{dirTransport}(T^{\rightsquigarrow}(\varphi)) : T(a) \xrightarrow{+} T(b)$. If $S : A \xrightarrow{-} \mathcal{U}_k$ is contravariant, we write $\varphi^* :\equiv \mathsf{dirTransport}(T^{\rightsquigarrow}(\varphi)) : S(b) \xrightarrow{+} S(a)$.

Remark 3.3.10. One can show that dirTransport preserves category structure. In section 3.8.9, the directed univalence axiom will turn dirTransport into an equivalence: $(A \rightsquigarrow_{\mathcal{U}_k} B) \overset{+}{\simeq} (A \xrightarrow{+} B)$. Then we can say that a morphism between types is the same as a function between types.

When we regard morphisms as a generalization of functions, there is more to say. We define a type of isomorphisms:

Definition 3.3.11. Let $A :^{+} \mathcal{U}_k$ be a type, take $a, b :^{\times} A$ and $\varphi :^{\times} a \rightsquigarrow_A b$. We define the types of **left and right inverses** of φ as follows:

$$\mathsf{leftInv}(\varphi) :\equiv \sum_{\lambda:b\to a}^{+} (\lambda \circ \varphi) = \mathsf{id}\, a \qquad \mathsf{rightInv}(\varphi) :\equiv \sum_{\rho:b\to a}^{+} (\varphi \circ \rho) = \mathsf{id}\, b. \tag{3.41}$$

We call φ an **isomoprhism** if it has a left and a right inverse:

$$\mathsf{isIsom}(\varphi) :\equiv \mathsf{leftInv}(\varphi) \times \mathsf{rightInv}(\varphi). \tag{3.42}$$

The type of **isomorphisms** from a to b is defined:

$$a \cong_A b :\equiv \sum_{\varphi:a\rightsquigarrow b}^{+} \mathsf{isIsom}(\varphi). \tag{3.43}$$

We get another generalization of the transport lemma:

Lemma 3.3.12 (Categorical transport)**.** For any type $A :^{=} \mathcal{U}_k$, equal objects of A are isomorphic:

$$\mathsf{catTransport} : \prod_{a,b:A}^{=} (a =_A b) \xrightarrow{+} (a \cong_A b). \tag{3.44}$$

Proof. By path induction, we need only prove $\prod_{a:A}^{=} a \cong_A a$, which is proven by the identity isomorphism $(\mathsf{id}\, a, \mathsf{id}\, a, \mathsf{refl}(\mathsf{id}\, a), \mathsf{refl}(\mathsf{id}\, a))$, which is indeed isovariant in A and a. □

Remark 3.3.13. One can show that catTransport preserves groupoid structure. In section 3.9, the categorical univalence axiom will turn catTransport into an equivalence: $(a =_A b) \overset{+}{\simeq} (a \cong_A b)$. Then we can say that a path is the same as an isomorphism.

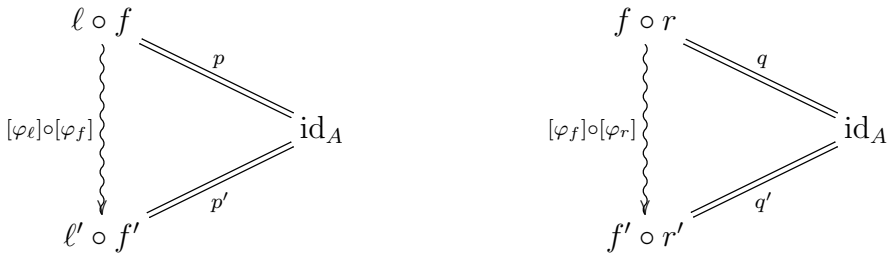
Finally, we have the following result:

Theorem 3.3.14. A morphism between equivalences or isomorphisms, is always an isomorphism:

$$\prod_{A,B:\mathcal{U}_k}^{=} \prod_{d,e:A\overset{+}{\simeq}B}^{=} (d \rightsquigarrow e) \xrightarrow{+} (d \cong e), \qquad \prod_{a,b:A}^{=} \prod_{\zeta,\eta:a\cong b}^{=} (\zeta \rightsquigarrow \eta) \xrightarrow{+} (\zeta \cong \eta). \tag{3.45}$$

Sketch of proof. We only sketch a proof for equivalences, but the other one is completely analogous.

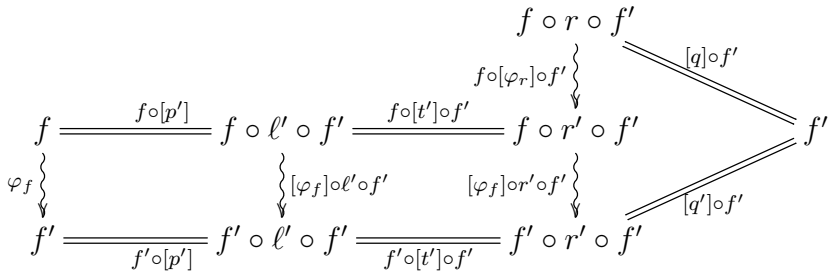
By (dependent) pair induction, we may assume that $d \equiv (f, \ell, p, r, q)$ and $e \equiv (f', \ell', p', r', q')$. A morphism $\varphi : d \rightsquigarrow e$ implies morphisms $\varphi_f : f \rightsquigarrow f'$, $\varphi_\ell : \ell \rightsquigarrow \ell'$ and $\varphi_r : r \rightsquigarrow r'$. Since identity types are groupoids, we also know that p equals p' and q equals q' after 'transporting'. In the following sections, we will see that this means that the following diagrams commute:



(3.46)

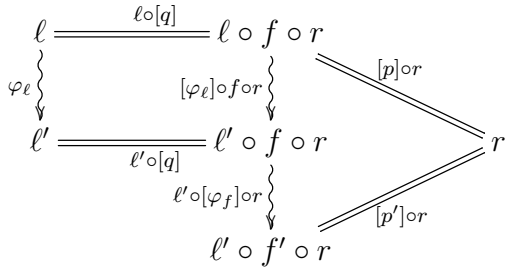
In addition, we know (from an earlier proof) $t : \ell = r$ and $t' : \ell' = r'$. We show that φ_f, φ_ℓ and φ_r are isomorphisms by constructing commutative diagrams where

we can walk from the target to the source. For φ_f, we can take



(3.47)

The square diagrams commute because morphisms/paths in the left argument of $\circ$ commute with morphisms/paths in the right argument (see Remark 3.6.8). The commutativity of the triangular one is proven by φ_q. Our morphism φ_f is on the utter left, and we can walk from f' to f along the outskirts. For φ_ℓ, we can take



(3.48)

The diagram for φ_r is analogous to this one. Thus, we have a componentwise isomorphism, which yields an isomorphism $d \cong e$. □

This indicates that the equivalence and isomorphism types *should* be bridged groupoids. Since at this point, the theory contains no rules for creating covariant functions simply by proving isovariance (it is highly non-trivial what such a rule should look like), we cannot prove that it is. In fact, since we are planning to identify equivalence, isomorphism and identity types by introducing univalence axioms, we should hope that the equality and isomorphism types are even bridgeless groupoids, as we can prove this for the identity type.

3.4 Type families are fibrations

3.4.1 In symmetric HoTT

In section 3.2, we showed that non-dependent functions $f : A \to C$ preserve paths. But what about dependent functions? Suppose we have a function $f : \prod_{x:A} C(x)$, elements $a, b : A$ and a path $p : a =_A b$. What can we say about $f(a) : C(a)$ and $f(b) : C(b)$? Surely not that they are propositionally equal, as they do not live in the same type. Of course $C : A \to \mathcal{U}_k$ is a non-dependent function, so we do know $C^=(p) : C(a) = C(b)$, but if we're going to put $f(a)$ in $C(b)$ just like that, we need $C(a) \equiv C(b)$.

At the very least, we can prove that $(a, f(a)) = (b, f(b))$ in $\sum_{x:A} C(x)$, since $g :\equiv x \mapsto (x, f(x))$ is a non-dependent function. Let's look at this from a topological perspective. If a and b are not judgementally equal, then they are *different* points in the space A.

Furthermore, for every point $x \in A$, we have a space $C(x)$. The space $\sum_{x:A} C(x)$ can be regarded as the disjoint union of all these spaces $C(x)$. But then it turns out that when there is a path $p : a = b$, there happens to be a path $g^{=}(p) : (a, f(a)) = (b, f(b))$ across different spaces $C(x)$. So the union $\sum_{x:A} C(x)$, although disjoint, is not disconnected. We find that type families do not carry fibrewise topologies, but a single topology for the entire family.

In topology, a fibration $\pi : S \to A$ is a continuous function that has the path lifting property: given a point $s \in S$ and a path p from $\pi(s) \in A$ to a point $b \in A$, we can lift it to a path q from $s \in S$ to some point $t \in S$, so that π maps q to p. In particular, t is in the fibre of b. We can show this in HoTT:

Lemma 3.4.1 (Path lifting property)**.** Given a type family $C : A \to \mathcal{U}_k$, elements $a, b : A$ and $s : C(a)$ and a path $p : a =_A b$, there is a point $t : C(b)$ and path $q : (a, s) = (b, t)$ so that $\mathsf{prl}^{=}(q) = p$.

Proof. For t, we can take $p_*(s)$. By path induction, we can assume that $a \equiv b$ and $p \equiv \mathsf{refl}\, a$. Then $(b, p_*(s)) \equiv (a, s)$, so we may take $q :\equiv \mathsf{refl}\, (a, s)$. □

So we should understand type families as fibrations of their disjoint union, which we will call the **total space**. This is visualized in fig. 3.1. There, A is a topological space that can

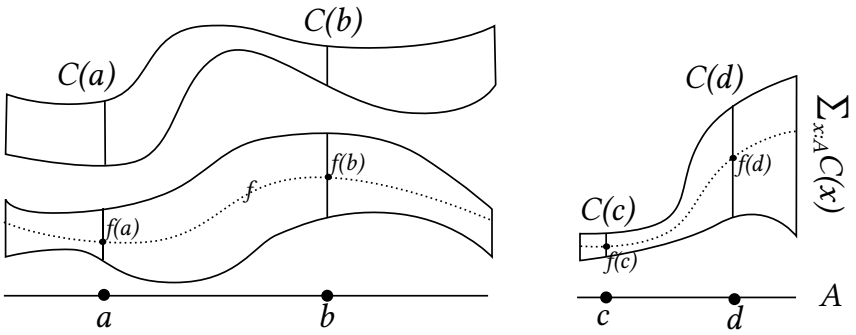


Figure 3.1: The type family $C : A \to \mathcal{U}_k$ interpreted as a fibration of $\sum_{x:A} C(x)$.

be embedded in $\mathbb{R}$, every fibre $C(x)$ can also be embedded in $\mathbb{R}$ (this has to do with the geometrical restrictions of a sheet of paper) and the total space $\sum_{x:A} C(x)$ is represented as a subset of $\mathbb{R}^2$. The types $C(x)$ are the fibres of the function $\mathsf{prl} : \sum_{x:A} C(x) \to A$, for example $C(b)$ is $\mathsf{prl}^{-1}(b)$. Whereas $\sum_{x:A} C(x)$ is the union of all the fibres, $\prod_{x:A} C(x)$ is the type of sections of prl. Indeed, a function $f : \prod_{x:A} C(x)$ gives rise to a function $g :\equiv x \mapsto (x, f(x)) : A \to \sum_{x:A} C(x)$ so that $\mathsf{prl} \circ g = \mathrm{id}_A$.

Not every function $S \to A$ is a fibration, however. Type families have the special property that they preserve equality, so that $a =_A b$ implies $C(a) = C(b)$ and via the transport function $C(a) \simeq C(b)$. In fig. 3.1, this is the case: the points a and b are connected, and $C(a)$ and $C(b)$ are clearly homotopically equivalent. Likewise, c and d are connected and $C(c) \simeq C(d)$. The fibres $C(a)$ and $C(c)$ are not equivalent, as $C(c)$ is connected and $C(a)$ is not, and they needn't be, as a and c are not connected.

We critically used the transport function in proving that type families respect the path lifting property. Topologically, we can imagine functions $\pi : S \to A$ for which the inverse images $\pi^{-1}(x)$ do not depend continuously on x (i.e. are not equivalent for connected

x), and which do not satisfy the path lifting property. Consider the example in fig. 3.2. Here, we have a space D consisting of 3 connected components and a function h from D

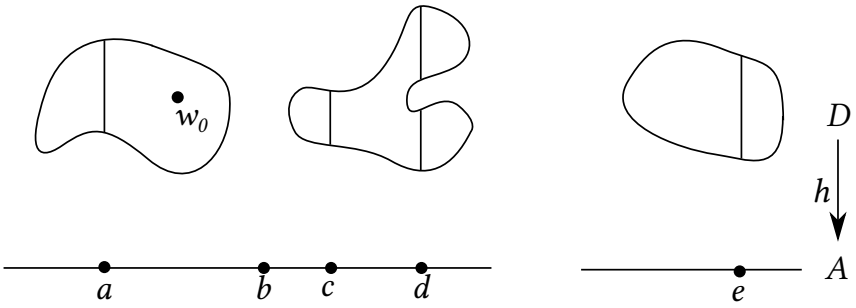


Figure 3.2: The function $h : D \to A$ fibrates D.

to a space A with two connected components. The points $a, b, c, d : A$ are connected, but their inverse images are not all homotopically equivalent. The inverse images of a and c are equivalent, but the one for b is empty, and the one for d is not connected.

A naive way to associate to this function a type family $C : A \to \mathcal{U}_k$ is by writing down the definition of inverse image: the inverse image of x is the set of all $w : D$ for which $h(w) = x$, so we could define

$$C :\equiv x \mapsto \sum_{w:D} h(w) =_A x : A \to \mathcal{U}_k. \tag{3.49}$$

But here, it is crucial to remember that the topological interpretation of a proof $p : h(w) = x$ is a *path* from $h(w)$ to x. Now, there is a path from $h(w_0)$ to a, b, c and d, so we will have copies of w_0 in $C(a), C(b), C(c)$ and $C(d)$. This certifies that the type C preserves equality, but it also severely distorts the original set-up. In fact, $C(a), C(b), C(c)$ and $C(d)$ will have two elements up to propositional equality, and $C(e)$ will have a single one, so the resulting fibration looks more like the one in the previous example, then like the original situation.

We try to capture some of these ideas in type theoretical results. First of all, we saw that the types $C(x)$ defined by (3.49), do not generally correspond to the inverse images $h^{-1}(x)$. But if h is actually a fibration, i.e. the projection from the total space of a type family, this *is* the case:

Lemma 3.4.2. Let A be a type and $C : A \to \mathcal{U}_k$ a type family. Then

$$\prod_{a:A} \left(\sum_{y:\sum_{x:A} C(x)} (\mathsf{prl}\, y = a) \right) \simeq C(a). \tag{3.50}$$

Proof. ($\leftarrow$) We can map $c : C(a)$ to $((a, c), \mathsf{refl}\, a)$. This is type correct, because $a : A$, $c : C(a)$ and $\mathsf{refl}\, a : \mathsf{prl}(a, c) = a$.

($\rightarrow$) After currying, we need to prove

$$\prod_{a,b:A} \prod_{c:C(b)} (\mathsf{prl}(b, c) = a) \to C(a). \tag{3.51}$$

Now $\mathsf{prl}(b, c)$ computes to b, so we can apply path induction and need to prove $\prod_{a:A} \prod_{c:C(a)} C(a)$ which is trivial. It computes as follows: $((a, c), \mathsf{refl}\, a)$ is mapped to c. This is clearly the inverse of the other arrow. □

We observed that elements of different fibres in a type family, can still be connected by a path. If we have $p : a =_A b$, then $f(a)$ and $f(b)$ are connected in the total space (the union of all fibres) not just by any path, but by a path that projects back to p. Thus, we can say that $f(a)$ and $f(b)$ are equal along p. We try to define a type of paths along p.

If we have a dependent function $f : \prod_{a:A} C(a)$ and we want to prove something about $f(a)$ and $f(b)$, given that $p : a =_A b$, then path induction allows us to assume that $a \equiv b$, $p \equiv \mathsf{refl}\, a$ and $f(a) \equiv f(b)$. Thus, if we want to prove something about points $c : C(a)$ and $c' : C(b)$ from the knowledge that c and c' are connected along p, then we should be able to assume precisely the same things. This suggests the following inductive definition for the identity type along p:

Inductive type family 3.4.3. Let $A : \mathcal{U}_k$ be a type and $C : A \to \mathcal{U}_k$ a type family over A. We define the **identity-along-identity type family** with constructor:

- $\frac{\mathsf{refl}\,\square}{\mathsf{refl}\,\square} : \prod_{a:A} \prod_{c:C(a)} \frac{c =_C c}{\mathsf{refl}\, a : a =_A a}$.

That is, $\frac{\mathsf{refl}\, c}{\mathsf{refl}\, a} : \frac{c =_C c}{\mathsf{refl}\, a : a =_A a}$. We should regard both $\frac{\mathsf{refl}\,\square}{\mathsf{refl}\,\square}$ and $\frac{\square =_\square \square}{\square : \square =_\square \square}$ as formal symbols. In general, we can read $\frac{c =_C c'}{p : a =_A b}$ as '$c : C(a)$ and $c' : C(b)$ are equal along p in the type family C'. The elements of this type are paths from c to c' in the total space, that are projected to p. We will not always mention the endpoints of p explicitly.

The induction principle says that, in order to prove something from $q : \frac{c =_C c'}{p : a =_A b}$, we may assume that $a \equiv b$, $p \equiv \mathsf{refl}\, a$, $c \equiv c'$ and $q \equiv \frac{\mathsf{refl}\, c}{\mathsf{refl}\, a}$.

$$\mathsf{ind}_{=_C/=_A} : \prod_{D:\ldots} \left(\prod_{a:A} \prod_{c:C(a)} D(a, a, \mathsf{refl}\, a, c, c, \mathsf{refl}\, c / \mathsf{refl}\, a) \right) \to \left(\prod_{a,b:A} \prod_{p:a=b} \prod_{c:C(a)} \prod_{c':C(b)} \prod_{q:c=_C c'/p} D(a, b, p, c, c', q) \right), \tag{3.52}$$

If we don't care about q, this is precisely what we need for function values of propositionally equal arguments. We will use this type as a tool to find out if other characterizations of equality along a path in more concrete situations, are correct. If they are, they should be equivalent to this type.

Heterogeneous equality types that express equality of elements from different types, are used more often in the literature [AMS07], and are available in the proof assistant Coq. Most often, however, there is one type for all paths q from c to c', regardless of the path p they project to.

The following lemma shows that dependent functions do preserve paths:

Lemma 3.4.4. For every type A, every type family $C : A \to \mathcal{U}_k$ and every function

$f : \prod_{x:A} C(x)$, we have

$$f^{=} : \prod_{a,b:A} \prod_{p:a=b} \frac{f(a) =_C f(b)}{p : a =_A b}. \tag{3.53}$$

Proof. After applying path induction to p, we only need reflexivity of $f(a)$ along reflexivity of a. □

The HoTTbook [Uni13] does not use an inductive heterogeneous identity type, as paths along a path are easily and fully generally characterized in symmetric HoTT:

Lemma 3.4.5. A path from $c : C(a)$ to $c' : C(b)$ along $p : a = b$ is the same as a path from $p_*(c)$ to c':

$$\prod_{a,b:A} \prod_{p:a=b} \prod_{c:C(a)} \prod_{c':C(b)} \prod_{q:c=_C c'/p} \frac{c =_C c'}{p : a =_A b} \simeq \left(p_*(c) =_{C(b)} c'\right). \tag{3.54}$$

Proof. ($\to$) By path-along-a-path induction, we need only prove

$$\prod_{a:A} \prod_{c:C(a)} (\mathsf{refl}\, a)_*(c) =_{C(a)} c, \tag{3.55}$$

which is proven by reflexivity. So we are mapping $a, a, \mathsf{refl}\, a, c, c, \frac{\mathsf{refl}\, c}{\mathsf{refl}\, a}$ to $\mathsf{refl}\, c$.

($\leftarrow$) By path induction, we need only prove

$$\prod_{a:A} \prod_{c,c':C(a)} c =_{C(a)} c' \to \frac{c =_C c'}{\mathsf{refl}\, a : a =_A a}. \tag{3.56}$$

After applying path induction again, we can use reflexivity along reflexivity. So we are mapping $a, a, \mathsf{refl}\, a, c, c, \mathsf{refl}\, c$ to $\mathsf{refl}\, c/\mathsf{refl}\, a$. This is clearly the inverse of the other arrow. □

3.4.2 In directed HoTT

The directed case is exciting because here, we can relax the continuity condition: type families still preserve paths, but objects in A may be connected by only a directed path (morphism), which invariant type families need not respect. Thus, our fibres may change fundamentally as we progress along a directed path. There are four (overlapping) cases, viewed in fig. 3.3.

In each case, we have a directed path (morphism) from a to d, passing through the points b and c. The type family C in the upper left, which is isovariant, maps the morphism to an equality, so that all the fibres along the directed path must be equivalent. This is similar to the situation in the symmetric case, only we are now moving along a directed path in A.

The type family D in the lower left, which is covariant, maps morphisms to functions. So whenever a point y lies beyond x on the directed path, there must be a function $C(x) \to C(y)$. So there is a way to walk to the right through the fibration, but not to the left. This is observed in the figure: when we take a point in the $C(b)$ and walk to the right, we can only end up in one connected component of $C(c)$, i.e.

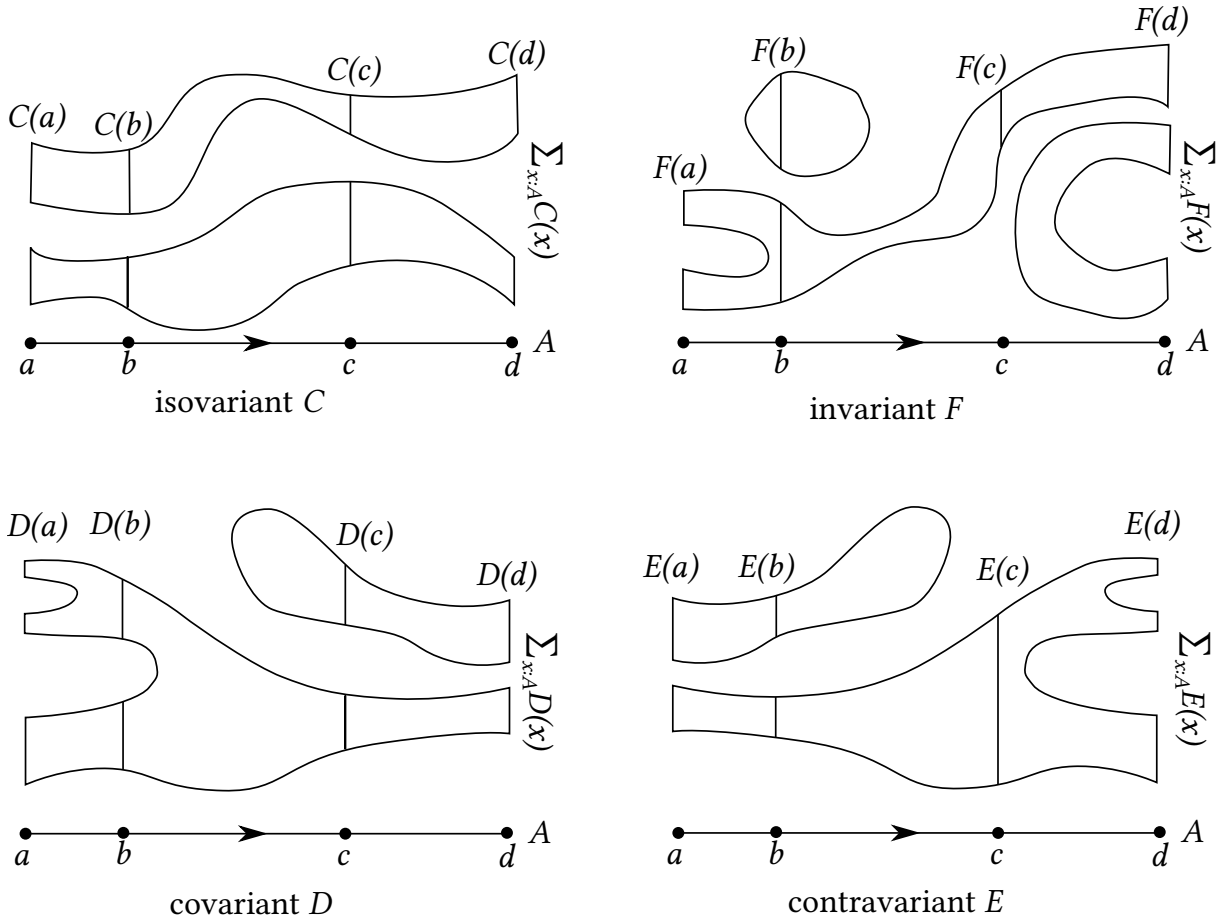


Figure 3.3: Behaviour of v-variant type families along a morphism.

the function value is unique up to propositional equality. Conversely, when we take a point in the upper connected component of $C(c)$ and walk to the left, we reach a dead end, because those points are not in the image of the function $C(b) \to C(c)$. When we take a point in the lower component and walk to the left, the component of $C(b)$ where we end up, is not uniquely determined. This indicates that the function is not injective.

The type family E in the lower right is contravariant, so the situation is flipped horizontally.

The type family F in the upper right is invariant, so we can get any kind of fibration. In the case shown in the figure, there is no unique way to walk in either direction. However, there is still a notion of equality across the type family. Indeed, all points of $F(a)$ are equal to the points in the upper component of $F(d)$ along the directed path. This is an interesting observation: **invariant type families F map morphisms $a \rightsquigarrow_A d$ to binary relations between $F(a)$ and $F(d)$.** There are a few remarks to be made about this. First of all, in our examples the total space is always a groupoid: there are no non-symmetric paths. If we allowed those, then we get something more bizarre than a binary relation: given $\alpha : F(a)$ and $\delta : F(d)$, we may speak of morphisms $\alpha \rightsquigarrow \delta$ and $\delta \rightsquigarrow \alpha$, as well as paths $\alpha = \delta$.

Secondly, this conclusion is based on an informal topological argument. However, we will derive some type theoretical results in a moment that reflect this observation, and the topological view provides a frame to think about these results.

It should be emphasized that type families of any variance still map paths to equivalences.

A question we face is: what is the total space? In the symmetric case, it was simply the Σ-type, but in directed HoTT, we have Σ-types of all variances. For the covariant sum, we will see that equality of pairs means componentwise equality, and a morphism between pairs is a pointwise morphism. This means that if we take the covariant sums in the four cases in fig. 3.3, then all symmetric paths remain within their fibre, and all directed paths go to the right.

The covariant sum will sometimes be useful, but it is not the total space one would expect when looking at fig. 3.3. In the contravariant sum, the symmetric paths are the same, but the directed paths go to the left. In the invariant sum, even directed paths have to stay in their fibre.

The isovariant sum is more interesting: here, (a, c) and (b, c') are equal if (but not only if) there is a morphism $a \rightsquigarrow b$ and c and c' are equal along that morphism. There is a morphism from (a, c) to (b, c') if (but not only if) there is a morphism in either direction between a and b, and a morphism from c to c' in the total space. We shall call the isovariant sum $\sum^{=}_{a:A} C(a)$ the **total space**. Unfortunately, the function that fibrates this total space cannot be given in type theory, as it is not functorial: it does not preserve paths along a morphism. If we want to be able to project back, we need to use the covariant sum:

Lemma 3.4.6. Let $A :^{=} \mathcal{U}_k$ be a type and $C :^{=} A \to \mathcal{U}_k$ a type family. Then

$$\prod_{a:A}^{+} \left(\sum_{y:\sum^{+}_{x:A} C(x)}^{+} (\mathsf{prl}\, y =_A a) \right) \overset{+}{\simeq} C(a). \tag{3.57}$$

Proof. ($\leftarrow$) We can map $c :^{+} C(a)$ to $((a, c), \mathsf{refl}\, a)$. This is type correct, because $a :^{+} A$, $c :^{+} C(a)$ and $\mathsf{refl}\, a :^{+} \mathsf{prl}(a, c) = a$.

($\rightarrow$) After currying, we need to prove

$$\prod_{a,b:A}^{+} \prod_{c:C(b)}^{+} (\mathsf{prl}(b, c) = a) \to C(a). \tag{3.58}$$

Now $\mathsf{prl}(b, c)$ computes to b, so we can apply path induction (which would even give a function isovariant in a and b) and need to prove $\prod^{=}_{a:A} \prod^{+}_{c:C(a)} C(a)$ which is proven by the identity function $\mathrm{id}_{C(a)}$ which happens to be isovariant in a. This proof computes as follows: $((a, c), \mathsf{refl}\, a)$ is mapped to c. This is clearly the inverse of the other arrow. □

For contravariant E, the projection $\mathsf{prl} : \left(\sum^{+}_{a:A} E(a)\right) \overset{+}{\to} A$ is apparently a Grothendieck fibration:

Definition 3.4.7 (Grothendieck fibration)**.** Let $\mathcal{S}$ and $\mathcal{A}$ be categories and $\pi : \mathcal{S} \to \mathcal{A}$ a covariant functor. A morphism $\sigma \in \mathrm{Hom}_{\mathcal{S}}(s,t)$ is called **cartesian** (with respect to π) if, for any morphism $\rho \in \mathrm{Hom}_{\mathcal{S}}(r,t)$, we can lift any factorization $\pi(\rho) = \pi(\sigma) \circ \alpha$

$$\begin{array}{ccccc} \pi(s) & & \xleftarrow{\quad\alpha\quad} & & \pi(r) \\ & {\scriptstyle \pi(\sigma)} \searrow & & \swarrow {\scriptstyle \pi(\rho)} & \\ & & \pi(t), & & \end{array} \tag{3.59}$$

to a factorization $\rho = \sigma \circ \psi$ such that $\pi(\psi) = \alpha$.

The functor π is called a **Grothendieck fibration** if, for any $t \in \mathcal{S}$, we can lift any morphism $\beta \in \mathrm{Hom}_{\mathcal{A}}(b, \pi(t))$ to a cartesian morphism $\sigma \in \mathrm{Hom}_{\mathcal{S}}(s,t)$ so that $\pi(\sigma) = \beta$.

By reverting all arrows, we arrive at the notion of a **cocartesian** morphism and a **Grothendieck opfibration**. [nLa15][Gro64]

Lemma 3.4.8 (Cartesian morphism lifting property)**.** Given a type $A :^{=} \mathcal{U}_k$, a *contravariant* type family $C :^{=} A \xrightarrow{-} \mathcal{U}_k$, elements $b, c :^{=} A$ and $t :^{+} C(c)$, we can lift a morphism $\beta :^{+} b \rightsquigarrow_A c$ to $\sigma : (b,s) \rightsquigarrow (c,t)$ such that $\mathsf{prl}^{\rightsquigarrow}(\sigma) = \beta$.

Moreover, this morphism is cartesian: for all $a :^{=} A$, $r :^{=} C(a)$ and $\rho :^{+} (a,r) \rightsquigarrow (c,t)$, we can lift every $\alpha :^{+} a \rightsquigarrow_A b$ that factorizes $\mathsf{prl}^{\rightsquigarrow}(\rho) = \mathsf{prl}^{\rightsquigarrow}(\sigma) \circ \alpha$, to a morphism $\psi : (a,r) \rightsquigarrow (b,s)$ that factorizes $\rho = \sigma \circ \psi$ so that $\mathsf{prl}^{\rightsquigarrow}(\psi) = \alpha$.

It is interesting to apply this lemma visually to each of the four drawings, and not just for the points a, b and c written there. There are two reasons why the lemma fails in the covariant and invariant cases. Firstly, the dead end caused by non-surjectivity disables you to lift morphisms. Secondly, in components that have no dead ends, the forking to the left caused by non-injectivity, disproves that there is always a cartesian lifting.

Proof. We have a transport function $\varphi^* : C(c) \xrightarrow{+} C(b)$, so we can take $s :\equiv \varphi^*(t)$. We now have to prove:

$$\prod_{b,c:A}^{=} \prod_{\beta:b\rightsquigarrow c}^{+} \prod_{t:C(c)}^{+} (b, \varphi^*(t)) \rightsquigarrow (c,t). \tag{3.60}$$

The variance of the type on the right allows morphism induction on β. Then we can assume that $b \equiv c$ and $\varphi^*(t)$ computes to t, so we can take $\sigma :\equiv \mathsf{id}(c,t)$. As identity projects to identity, this is indeed a lifting.

Using target-based morphism induction on β (a and c are fixed as we are dealing with paths in $a \rightsquigarrow c$), both β and σ become identity morphisms. Then the fact that α factorizes $\mathsf{prl}^{\rightsquigarrow}(\rho)$ over β means that it equals $\mathsf{prl}^{\rightsquigarrow}(\rho)$, so we can take $\psi :\equiv \rho$. □

For covariant C, the function $\mathsf{prl} : \left(\sum_{a:A}^{+} C(a)\right) \xrightarrow{+} A$ is a Grothendieck opfibration.

Observe that the lemmas in section 3.2 about functor behaviour of functions, are all isovariant in the endpoints of paths and morphisms, as well as in the relevant function. So when we want to state analogues for dependent functions, the variances are likely to be the same. Suppose that given $f :^{=} \prod^{v}_{a:A} C(a)$, points $a, b :^{=} A$ and a path $p :^{+} a =_A b$, we want to say something about $f(a)$ and $f(b)$. Then by path induction, we may assume that $a \equiv b$, $p \equiv \mathsf{refl}\, a$ and $f(a) \equiv f(b)$. Moreover, since both f and a are available isovariantly, so will $f(a)$ be. This leads to the following definition for identity along identity:

Inductive type family 3.4.9. Let $A : \mathcal{U}_k$ be a type and $C : A \xrightarrow{u} \mathcal{U}_k$ a type family over A. We define the **identity-along-identity type family**, which is covariant in A and C, with constructor:

- $\frac{\mathsf{refl}\,\square}{\mathsf{refl}\,\square} : \prod^{=}_{a:A} \prod^{=}_{c:C(a)} \frac{c =_C c}{\mathsf{refl}\, a =_{a=_A a} a}$.

The v-variant induction principle says that, in order to prove something from $q :^{v} \frac{c =_C c'}{p:a=_A b}$, we may assume that $a \equiv b$, $p \equiv \mathsf{refl}\, a$, $c \equiv c'$ and $q \equiv \frac{\mathsf{refl}\, c}{\mathsf{refl}\, a}$, where a and c are available isovariantly.

$$\mathsf{ind}^{v}_{=_C/=_A} : \prod^{=}_{D:\dots} \left(\prod^{=}_{a:A} \prod^{=}_{c:C(a)} D(a, a, \mathsf{refl}\, a, c, c, \mathsf{refl}\, c/\mathsf{refl}\, a) \right) \\ \to \left(\prod^{=}_{a,b:A} \prod^{v}_{p:a=b} \prod^{=}_{c:C(a)} \prod^{=}_{c':C(b)} \prod^{v}_{q:c=_C c'/p} D(a, b, p, c, c', q) \right). \tag{3.61}$$

If we don't care about q, this is precisely what we need for function values of propositionally equal arguments.

Dependent functions still preserve paths:

Lemma 3.4.10. For every type $A :^{=} \mathcal{U}_k$, every type family $C :^{=} A \xrightarrow{u} \mathcal{U}_k$ and every function $f :^{=} \prod^{v}_{x:A} C(x)$, we have

$$f^{=} : \prod^{=}_{a,b:A} \prod^{+}_{p:a=b} \frac{f(a) =_C f(b)}{p : a =_A b}. \tag{3.62}$$

Proof. After applying path induction to p, we only need reflexivity of $f(a)$ along reflexivity of a. □

Paths along paths are still very easy to characterize:

Lemma 3.4.11. A path from $c : C(a)$ to $c' : C(b)$ along $p : a = b$ is the same as a path from $p_*(c)$ to c':

$$\prod_{a,b:A}^{=} \prod_{p:a=b}^{+} \prod_{c:C(a)}^{=} \prod_{c':C(b)}^{=} \prod_{q:c=_C c'/p}^{+} \frac{c =_C c'}{p : a =_A b} \simeq \left(p_*(c) =_{C(b)} c'\right). \tag{3.63}$$

Proof. ($\to$) By path-along-a-path induction, we need only prove

$$\prod_{a:A}^{=} \prod_{c:C(a)}^{=} (\mathsf{refl}\, a)_*(c) =_{C(a)} c', \tag{3.64}$$

which is proven by reflexivity. So we are mapping $a, a, \mathsf{refl}\, a, c, c, \frac{\mathsf{refl}\, c}{\mathsf{refl}\, a}$ to $\mathsf{refl}\, c$.

($\leftarrow$) By path induction, we need only prove

$$\prod_{a:A}^{=} \prod_{c,c':C(a)}^{=} c =_{C(a)} c' \overset{+}{\to} \frac{c =_C c'}{\mathsf{refl}\, a : a =_A a}. \tag{3.65}$$

After applying path induction again, we can use reflexivity along reflexivity. So we are mapping $a, a, \mathsf{refl}\, a, c, c, \mathsf{refl}\, c$ to $\mathsf{refl}\, c/\mathsf{refl}\, a$. This is clearly the inverse of the other arrow. □

As any function maps paths to paths, it is not interesting to study morphisms along paths. However, it is interesting to study paths and morphisms (in either direction) along morphisms. Therefore, we need the following types:

$$\frac{c =_C c'}{\varphi : a \rightsquigarrow_A b}, \qquad \frac{c \rightsquigarrow_C c'}{\varphi : a \rightsquigarrow_A b}, \qquad \frac{c \leftsquigarrow_C c'}{\varphi : a \rightsquigarrow_A b}. \tag{3.66}$$

We give them a completely analogous definition, only with different variance in their indices. All three types are contravariant in a and covariant in b. The second one is contravariant in c and covariant in c', the third one is contravariant in c and covariant in c'. The induction principles will require type families that have according variance. The constructor of the first type will be called $\frac{\mathsf{refl}\,\square}{\mathsf{id}\,\square}$; the ones for the other two will be called $\frac{\mathsf{id}\,\square}{\mathsf{id}\,\square}$.

For type families C that are remotely well-behaved, there is again a general characterization. We give it only for equalities along morphisms, the other cases are proven analogously.

Lemma 3.4.12. Let $C :\equiv A \xrightarrow{-} A \xrightarrow{+} \mathcal{U}_k$ be a type family that is contravariant in one argument and covariant in the other. Suppose we have objects $a, b :\equiv A$ and a morphism $\varphi :^{+} a \rightsquigarrow_A b$. Then we get transport functions:

$$C(a,a) \xrightarrow{\varphi_*} C(a,b) \xleftarrow{\varphi^*} C(b,b). \tag{3.67}$$

A path from $c : C(a,a)$ to $c' : C(b,b)$ along a morphism $\varphi : a \rightsquigarrow_A b$ is the same as a path from $\varphi_*(c)$ to $\varphi^*(c')$:

$$\prod_{a,b:A}^{=} \prod_{\varphi:a\rightsquigarrow b}^{+} \prod_{c:C(a,a)}^{=} \prod_{c':C(b,b)}^{=} \frac{c =_{x\overset{\times}{\mapsto}C(x,x)} c'}{\varphi : a \rightsquigarrow_A b} \simeq \left(\varphi_*(c) =_{C(a,b)} \varphi^*(c')\right). \tag{3.68}$$

Proof. ($\to$) The type family

$$a \overset{-}{\mapsto} b \overset{+}{\mapsto} \varphi \overset{+}{\mapsto} c \overset{\times}{\mapsto} c' \overset{\times}{\mapsto} \left(\varphi_*(c) =_{C(a,b)} \varphi^*(c')\right) \tag{3.69}$$

has the proper variance to apply path-along-a-morphism induction. Then we need only prove

$$\prod_{a:A}^{=} \prod_{c:C(a)}^{=} (\mathsf{id}\, a)_*(c) =_{C(a,a)} (\mathsf{id}\, a)^*(c), \tag{3.70}$$

which is proven by reflexivity. So we are mapping $a, a, \mathsf{id}\, a, c, c, \frac{\mathsf{refl}\, c}{\mathsf{id}\, a}$ to $\mathsf{refl}\, c$.

($\leftarrow$) By morphism induction, we need only prove

$$\prod_{a:A}^{=} \prod_{c,c':C(a)}^{=} c =_{C(a)} c' \overset{+}{\to} \frac{c =_C c'}{\mathsf{id}\, a : a \rightsquigarrow_A a}. \tag{3.71}$$

After applying path induction, we can use reflexivity along identity. So we are mapping $a, a, \mathsf{id}\, a, c, c, \mathsf{refl}\, c$ to $\mathsf{refl}\, c/\mathsf{id}\, a$. This is clearly the inverse of the other arrow. □

Note that this lemma characterizes paths along morphisms for co-, contra- and iso-variant type families, as well as for type families that are constructed by combining these. The only remaining families to investigate are those that are invariant 'by nature'. Let us see how that can happen:

- By using $\mathsf{strip} : A \overset{\times}{\to} A^{\mathsf{core}}$ in the definition of a type family. However, A^{core} itself is not the universe, so we would have to compose with $A^{\mathsf{core}} \xrightarrow{+/-/\times} \mathcal{U}_k$, but such functions are constructed by core-induction from $(A \overset{\times}{\to} \mathcal{U}_k)$, so this is a non-example, as we needed a invariant type family to begin with.
- By actively weakening a better behaved type family, e.g. if C is covariant, we could define $D :\equiv a \overset{\times}{\mapsto} C(a)$. However, one can prove that $\prod_{a:A} D(x) \simeq \prod_{a:A} C(x)$, so that we need not consider this case.

- Type families that have complicated higher order variance that has to be weakened to invariance. Remember that $\Box^{\mathsf{op}}$ was invariant as it preserves morphisms contravariantly.
- The identity type is by nature invariant in its indices.

Identity types

We start with the latter case: if $a, a' :^{=} A$, $\varphi :^{+} a \rightsquigarrow_A b$, $f, g : A \overset{v}{\to} B$, $p : f(a) =_B g(a)$ and $p' : f(a') =_B g(a')$, then what is a path $q : \frac{p=p'}{\varphi : a \rightsquigarrow a'}$ in the family $x \overset{\times}{\mapsto} f(x) = g(x)$?

Let us first replace p and p' by morphisms and assume f and g covariant. Then we get $\chi : f(a) \rightsquigarrow_B g(a)$ and $\chi' : f(a') \rightsquigarrow_B g(a')$ and the question is: what is a path $\frac{\chi = \chi'}{\varphi : a \rightsquigarrow a'}$? But we know the answer, it is a path from $q : \varphi_*(\chi)$ to $\varphi^*(\chi')$. But transporting morphisms along morphisms is just composition, so we get

$$q : g^{\rightsquigarrow}(\varphi) \circ \chi = \chi' \circ f^{\rightsquigarrow}(\varphi), \tag{3.72}$$

i.e. the following diagram commutes:

$$\begin{array}{ccc} f(a) & \overset{f^{\rightsquigarrow}(\varphi)}{\rightsquigarrow} & f(a') \\ {\scriptstyle \chi}\big\downarrow & & \big\downarrow{\scriptstyle \chi'} \\ g(a) & \underset{g^{\rightsquigarrow}(\varphi)}{\rightsquigarrow} & g(a') \end{array} \tag{3.73}$$

As it happens, the situation is identical for paths. We only show this for covariant f and g.

Lemma 3.4.13. Suppose we have types $A, B :^{=} \mathcal{U}_k$ and functions $f, g : A \overset{+}{\to} B$. Define $C :\equiv x \overset{\times}{\mapsto} f(x) =_B g(x)$. A path from $p : C(a)$ to $p' : C(a')$ along a morphism $\varphi : a \rightsquigarrow_A a'$ is the same as a path $g^{\rightsquigarrow}(\varphi) \circ \mathsf{toMorph}(p) = \mathsf{toMorph}(p') \circ f^{\rightsquigarrow}(\varphi)$:

$$\prod_{a,a':A}^{=} \prod_{\varphi : a \rightsquigarrow a}^{+} \prod_{c:C(a)}^{=} \prod_{c':C(a')}^{=} \frac{p =_C p'}{\varphi : a \rightsquigarrow_A a'} \simeq \left(g^{\rightsquigarrow}(\varphi) \circ \mathsf{toMorph}(p) = \mathsf{toMorph}(p') \circ f^{\rightsquigarrow}(\varphi)\right). \tag{3.74}$$

In other words, it means the following diagram commutes:

$$\begin{array}{ccc} f(a) & \overset{f^{\rightsquigarrow}(\varphi)}{\rightsquigarrow} & f(a') \\ {\scriptstyle p}\big\| & & \big\|{\scriptstyle p'} \\ g(a) & \underset{g^{\rightsquigarrow}(\varphi)}{\rightsquigarrow} & g(a') \end{array} \tag{3.75}$$

Proof. ($\to$) The type family

$$a \overset{-}{\mapsto} a' \overset{+}{\mapsto} \varphi \overset{+}{\mapsto} p \overset{\times}{\mapsto} p' \overset{\times}{\mapsto} \left(g^{\rightsquigarrow}(\varphi) \circ \mathsf{toMorph}(p) =_{f(a)\rightsquigarrow g(a')} \mathsf{toMorph}(p') \circ f^{\rightsquigarrow}(\varphi)\right)$$

has the proper variance to apply path-along-a-morphism induction. Then we need only prove

$$\prod_{a:A}^{=} \prod_{p:C(a)}^{=} \left(g^{\rightsquigarrow}(\mathsf{id}\, a) \circ \mathsf{toMorph}(p) = \mathsf{toMorph}(p) \circ f^{\rightsquigarrow}(\mathsf{id}\, a)\right), \tag{3.76}$$

but both sides compute to $\mathsf{toMorph}(p)$, so we can use reflexivity. Then we are mapping $a, a, \mathsf{id}\, a, p, p, \frac{\mathsf{refl}\, p}{\mathsf{id}\, a}$ to $\mathsf{refl}\, \mathsf{toMorph}(p)$.

($\leftarrow$) By morphism induction, we need only prove

$$\prod_{a:A}^{=} \prod_{p,p':C(a)}^{=} \mathsf{toMorph}(p) =_{f(a)\rightsquigarrow g(a)} \mathsf{toMorph}(p') \overset{+}{\to} \frac{p =_C p'}{\mathsf{refl}\, a : a =_A a}. \tag{3.77}$$

Lemma 3.9.8 will assert that $(\mathsf{toMorph}(p) =_{f(a)\rightsquigarrow g(a)} \mathsf{toMorph}(p')) \simeq p =_{C(a)} p'$. Then after applying path induction, we need only prove

$$\prod_{a:A}^{=} \prod_{p:C(a)}^{=} \frac{p =_C p}{\mathsf{refl}\, a : a =_A a}, \tag{3.78}$$

which is proven by reflexivity along identity. So we are mapping a, a, $\mathsf{id}\, a$, p, p, $\mathsf{refl}\, \mathsf{toMorph}(p)$ to $\mathsf{refl}\, c/\mathsf{id}\, a$. This is clearly the inverse of the other arrow. □

We can now understand why refl had to be isovariant. Suppose we have $a, b : A$ and $\varphi : a \rightsquigarrow_A b$. Then we would expect that the following diagram commutes:

$$\begin{array}{ccc} a & \overset{\varphi}{\rightsquigarrow} & b \\ {\scriptstyle \mathsf{refl}\, a}\Big\| & & \Big\|{\scriptstyle \mathsf{refl}\, b} \\ a & \underset{\varphi}{\rightsquigarrow} & b \end{array} \tag{3.79}$$

which is the same as saying that $\mathsf{refl}\, a$ equals $\mathsf{refl}\, b$ in the type family $x \overset{\times}{\mapsto} (x =_A x)$.

The opposite type

As an example of a type family that is considered invariant because its higher variance could not be weakened to any other basic variance, we take the type family $X \overset{\times}{\mapsto} X^{\mathsf{op}}$. If we have a morphism $\varphi : a \rightsquigarrow_A b$ and a family $C : A \overset{-}{\to} A \overset{+}{\to} \mathcal{U}_k$, then we cannot immediately transport from $C(a,a)^{\mathsf{op}}$ and $C(b,b)^{\mathsf{op}}$ to $C(a,b)^{\mathsf{op}}$, as $\square^{\mathsf{op}}$ discards morphisms. However, we do have

$$C(a,a)^{\mathsf{op}} \xrightarrow{\mathsf{unflip}} C(a,a) \xrightarrow{\varphi_*} C(a,b) \xrightarrow{\mathsf{flip}} C(a,b)^{\mathsf{op}} \xleftarrow{\mathsf{flip}} C(a,b) \xleftarrow{\varphi^*} C(b,b) \xleftarrow{\mathsf{unflip}} C(b,b)^{\mathsf{op}},$$

which is basically the transport functions φ_* and φ^* transported along the flipping equivalence $X \bar{\simeq} X^{\mathsf{op}}$.

Now if we have a covariant function $f : \prod^{+}_{x:A} C(x,x)^{\mathsf{op}}$, we could expect a morphism $(\mathsf{flip} \circ \varphi_* \circ \mathsf{unflip})(f(a)) \rightsquigarrow_{C(a,b)^{\mathsf{op}}} (\mathsf{flip} \circ \varphi^* \circ \mathsf{unflip})(f(b))$. However, we cannot prove this from morphism induction on φ, because the identity on the right is invariant in a and b, which is the case because $C(a,b)^{\mathsf{op}}$ is invariant in a and b since $\square^{\mathsf{op}}$ is invariant. Had we not weakened the contravariant morphism induction principle to fit our type system, then we could have applied it here.

What we can do, however, is prove that there is a morphism $(\varphi^* \circ \mathsf{unflip})(f(b)) \rightsquigarrow_{C(a,b)} (\varphi_* \circ \mathsf{unflip})(f(a))$ (note the swapping of a and b). This is conceptually the same, because the idea of the opposite type is that a morphism $x \rightsquigarrow_X y$ is the same as a morphism $\mathsf{flip}\, y \rightsquigarrow_{X^{\mathsf{op}}} \mathsf{flip}\, x$. We prove that this is the right characterization of a morphism in the type family $x \overset{\times}{\mapsto} C(x,x)^{\mathsf{op}}$:

Lemma 3.4.14. A morphism from $c : C(a,a)^{\mathsf{op}}$ to $c' : C(b,b)^{\mathsf{op}}$ along a morphism $\varphi : a \rightsquigarrow_A b$ is the same as a morphism from $(\varphi^* \circ \mathsf{unflip})(c')$ to $(\varphi_* \circ \mathsf{unflip})(c)$:

$$\prod^{=}_{a,b:A} \prod^{+}_{\varphi:a\rightsquigarrow_b} \prod^{=}_{c:C(a,a)^{\mathsf{op}}} \prod^{=}_{c':C(b,b)^{\mathsf{op}}} \frac{c \rightsquigarrow_{C^{\mathsf{op}}(\square)} c'}{\varphi : a \rightsquigarrow_A b} \overset{+}{\simeq} \left((\varphi^* \circ \mathsf{unflip})(c') \rightsquigarrow_{C(a,b)} (\varphi_* \circ \mathsf{unflip})(c)\right).$$

Proof. ($\rightarrow$) The type family

$$a \overset{-}{\mapsto} b \overset{+}{\mapsto} \varphi \overset{+}{\mapsto} c \overset{-}{\mapsto} c' \overset{+}{\mapsto} \left((\varphi^* \circ \mathsf{unflip})(c') \rightsquigarrow_{C(a,b)} (\varphi_* \circ \mathsf{unflip})(c)\right) \tag{3.80}$$

has the proper variance to apply morphism-along-a-morphism induction. Then we need only prove

$$\prod^{=}_{a:A} \prod^{=}_{c:C(a)} (\mathsf{id}\, a)^*(c) \rightsquigarrow_{C(a,a)} (\mathsf{id}\, a)_*(c), \tag{3.81}$$

which is proven by the identity morphism. So we are mapping $a, a, \mathsf{id}\, a, c, c, \frac{\mathsf{id}\, c}{\mathsf{id}\, a}$ to $\mathsf{id}\, c$.

($\leftarrow$) By morphism induction on φ, we need only prove

$$\prod^{=}_{a:A} \prod^{=}_{c,c':C(a)} c' \rightsquigarrow_{C(a)} c \overset{+}{\to} \frac{c \rightsquigarrow_C c'}{\mathsf{id}\, a : a \rightsquigarrow_A a}. \tag{3.82}$$

After applying morphism induction again induction, we can use identity along identity. So we are mapping $a, a, \mathsf{id}\, a, c, c, \mathsf{id}, c$ to $\mathsf{id}\, c / \mathsf{id}\, a$. This is clearly the inverse of the other arrow. $\square$

3.5 Bridges

3.5.1 Discussion

Why do we need bridges?

The idea to start reasoning about bridges, comes from two observations that did not necessarily ask for a common answer. The first observation is that invariant type families map morphisms to relations, as demonstrated in the previous section. However, we know that the type of invariant type families $A \xrightarrow{\times} \mathcal{U}_k$ is covariantly equivalent to $A^{\mathsf{core}} \xrightarrow{+} \mathcal{U}_k$. This indicates that the category structure of A has left a trace on A^{core}. Bridgeless groupoids on the other hand, do not exhibit such behaviour. Indeed, we can define $C : \mathbb{N} \xrightarrow{+} \mathcal{U}_k$ by saying $C(\mathsf{zero}) :\equiv \mathsf{Fin}\,47$ and $C(\mathsf{succ}\,n) :\equiv (\mathbb{N} \simeq \mathbb{N} \times \mathbb{N})$, for all n. Now, there is absolutely no natural way of comparing elements of the finite type with 47 (interchangeable) elements to bijections between the naturals and the type of pairs of naturals.

Secondly, if it weren't for bridges, we would have put $= \circ \times \equiv \times$. But it turns out that this doesn't play well with the inference rule for extending the context:

$$\frac{\Gamma \vdash A :^v \mathcal{U}_k}{\Gamma, x :^= A \vdash \mathsf{Ctx}}\mathsf{extendCtx}. \tag{3.83}$$

Earlier, we derived the following rule from it:

$$\frac{\Gamma \vdash A :^v \mathcal{U}_k}{= \circ \Gamma, x :^= A \vdash \mathsf{Ctx}}\mathsf{extendCtx} \tag{3.84}$$

by first composing the premise with isovariance and then applying $\mathsf{extendCtx}$. If $= \circ \times \equiv =$, then $= \circ \Gamma \equiv \Gamma^=$. However, if we have $= \circ \times \equiv \times$, this no longer holds, meaning some (fairly important) functions cannot be created any more. For example, there is the path induction principle, which is defined from the path induction operator as follows:

$$\mathsf{ind}_{=_A} :\equiv C \overset{=}{\mapsto} \left(f : \prod_{x:A} C(x, x, \mathsf{refl}\,x) \right) \overset{+}{\mapsto} \left(a \overset{=}{\mapsto} b \overset{=}{\mapsto} p \overset{+}{\mapsto} \mathsf{ind}_{=_A}(C, x.f(x), a, b, p) \right).$$

To construct this function, we repeatedly apply the function definition rule (also called λ-abstraction, which turns an assumption into an argument), starting from the judgement

$$C :^= \cdots, f :^+ \cdots, a :^= A, b :^= A, p :^= a =_A b \vdash \mathsf{ind}_{=_A}(C, x.f(x), a, b, p) :^+ \cdots \tag{3.85}$$

where $\cdots$ replaces long and complicated types.

Now the problem is that we cannot even obtain this context, because at some point we have to add the identity type $a =_A b$ to the context, but if $\Gamma \vdash a =_A b : \mathcal{U}_k$, then Γ needs to assume $a, b :^\times A$. However, when we add the identity type to the context, we get $(= \circ \Gamma), p : a =_A b \vdash \mathsf{Ctx}$, but $(= \circ \Gamma)$ would still be assuming a and b invariantly, whereas we need them isovariantly.

One way to solve this, would be by introducing a fifth **constant** variance, which maps no information to equality. Then of course we would have $\mathrm{ct} \circ \times \equiv \mathrm{ct}$. By composing with constancy before applying $\mathsf{extendCtx}$, we could derive

$$\frac{\Gamma \vdash A :^{v} \mathcal{U}_k}{(\mathrm{ct} \circ \Gamma), x :^{\mathrm{ct}} A \vdash \mathsf{Ctx}}. \tag{3.86}$$

(The constancy of x is not derivable; it follows from a modification of $\mathsf{extendCtx}$ following the philosophy that we should take it in the strongest possible variance so that we can weaken it; the strongest possible variance is now constancy). However, Guideline 2.4.2 would then become: Asserting the sheer existence of a type or type family, falls under **constant** use. Then the constructor

$$\mathsf{inl} : \mathsf{Vec}_n\, A \xrightarrow{+} \mathsf{Vec}_n\, A + B \tag{3.87}$$

would depend constantly on n, meaning that the one for 8 is equal to the one for 65, but as 8 and 65 are completely unrelated, there is no obvious way to compare inl_8 to inl_{65}, leaving the claim meaningless. (Of course a function $f : \mathsf{Vec}_{65}\, A \xrightarrow{+} \mathsf{Vec}_8\, A$ would yield a commutative diagram, but this is just because $\mathsf{inl} : X \xrightarrow{+} X + B$ depends constantly on X.) Bridges allow us to distinguish between cases where it is meaningful to claim that terms are equal, and cases where it isn't, so they seem a natural way to address this situation.

What are they?

Let's look back at the invariant type family $F : A \to \mathcal{U}_k$ in fig. 3.3 on page 96. Note that any point in A has a neighbourhood where F is either co- or contravariant. This suggests that it may be sensible to think of a bridge $a \frown b$ as a so called **zigzag** of morphisms:

$$a \rightsquigarrow a_1 \leftsquigarrow a_2 \rightsquigarrow \ldots \leftsquigarrow a_{n-2} \rightsquigarrow a_{n-1} \leftsquigarrow b\,. \tag{3.88}$$

An isovariant function maps every one of these morphisms to equality, so the entire bridge is mapped to equality. A covariant function preserves the morphisms and a contravariant reverts all of them, so that both yield again a bridge. An invariant function maps each of the morphisms to a bridge, which we can tie together to obtain another bridge.

From this way of thinking, we can deduce the following properties of bridges:

- A morphism implies a bridge.
- In particular, there will be identity bridges.
- There is an anti-involution $\square^{\dagger} : (a \frown b) \to (b \frown a)$.
- Bridges can be composed.
- A bridge between types is the same as a relation.

To see the last property, take a bridge from A to D:

$$A \xrightarrow{f} B \xleftarrow{g} C \xrightarrow{h} D. \tag{3.89}$$

This constitutes a relation between A and D, where there is a morphism from $a : A$ to $d : D$ whenever there are $b : B$ and $c : C$ so that $(f(a) \rightsquigarrow_B b) \times (b \rightsquigarrow_B g(c)) \times (h(c) \rightsquigarrow_D d)$. Conversely, a relation between A and B should probably be defined as a diagram

$$A \xrightarrow{+} R \xleftarrow{+} B \tag{3.90}$$

which is indeed a bridge. (If we wanted to stay closer to the set theoretic definition of a relation, we might have opted for a diagram $A \leftarrow S \rightarrow B$. However, this definition does not give us transport functions.)

Of course in the core, where we have thrown away all morphisms, a bridge cannot possible 'be' a zigzag of morphisms, but to some extent they seem to behave as if they are.

It is quite possible though that this is not the best way to think of bridges. If we want to think more abstractly about them, then we have to be more careful about the action of functions of different variance on them. Our basic assumption is that invariant functions map morphisms to bridges. Suppose we have functions $f : A \xrightarrow{\times} B$ and $g : B \xrightarrow{v} C$. If g is co- or contravariant, then $g \circ f$ is invariant and hence maps morphisms to bridges. Since the only thing that is left in B from a morphism in A, is a bridge, this indicates that co- and contravariant functions should preserve bridges. However, for isovariant functions, there is no longer a good argument why they should map bridges to equality. So we should split up isovariance in strong isovariance (mapping bridges to equality) and weak isovariance (just mapping bridges to bridges). Guideline 2.4.2 will then state that asserting the sheer existence of a type, falls under strongly isovariant use. As most isovariant functions came forth from Guideline 2.4.2, we wouldn't often need weakly isovariant functions. Therefore, we proceed with the rule $= \circ \times \equiv =$.

3.5.2 Attempts to define bridges

It turns out to be quite difficult to find a good definition for a bridge type. A good bridge type should satisfy the following criteria:

1. Invariant functions map morphisms to bridges,
2. As a consequence, morphisms imply bridges (since the identity function can be weakened to invariance),
3. Isovariant functions map bridges to paths,
4. In the universe, a bridge between types is the same as a relation (it is probably sufficient that a bridge implies a relation, which is analogous to a transport lemma, whereas the other implication usually requires a univalence axiom),
5. There is an anti-involution $\square^\dagger : a \frown b \to b \frown a$ so that $(\mathsf{idbr}\, a)^\dagger = \mathsf{idbr}\, a$,

6. There is an identity bridge $\mathsf{idbr}\, a$ in every point a,

7. If bridges are composable, we should not be able to prove for every bridge β that $\beta^\dagger \circ \beta = \mathsf{idbr}$, as this does not hold in general for relations.

We make four attempts at defining bridges.

Using bivariance If a morphism $\varphi : a \rightsquigarrow_A b$ should imply a bridge $a \frown_A b$, then we should be able to prove this using the induction principle, which requires that $a \frown_A b$ is at least contravariant in a and covariant in b. However, a morphism $\chi : b \rightsquigarrow_A a$ should also imply a bridge $a \frown_A b$, so $a \frown_A b$ should also be covariant in a and contravariant in b. Actually this make sense: if bridges are zigzags, then we should be able to transport in either direction along a morphism $\psi : b \rightsquigarrow_A c$:

$$\psi_* : (a \frown_A b) \overset{+}{\leftrightarrows} (a \frown_A c). \tag{3.91}$$

This suggests that we should include **bivariant** functions, which map a morphism $a \rightsquigarrow b$ to morphisms $(f(a) \rightsquigarrow f(b)) \times (f(b) \rightsquigarrow f(a))$. If we apply this to a zigzag, then every segment gets mapped to morphisms in both directions, which we can compose. So a bivariant function will also map bridges $a \frown b$ to $(f(a) \rightsquigarrow f(b)) \times (f(b) \rightsquigarrow f(a))$. Finally, as any other function, it should preserve equality. We get the following composition table:

$+ \circ + \equiv +$	$+ \circ - \equiv -$	$+ \circ \times \equiv \times$	$+ \circ \mathrm{bi} \equiv \mathrm{bi}$	$+ \circ = \equiv =$
$- \circ + \equiv -$	$- \circ - \equiv +$	$- \circ \times \equiv \times$	$- \circ \mathrm{bi} \equiv \mathrm{bi}$	$- \circ = \equiv =$
$\times \circ + \equiv \times$	$\times \circ - \equiv \times$	$\times \circ \times \equiv \times$	$\times \circ \mathrm{bi} \equiv \times$	$\times \circ = \equiv =$
$\mathrm{bi} \circ + \equiv \mathrm{bi}$	$\mathrm{bi} \circ - \equiv \mathrm{bi}$	$\mathrm{bi} \circ \times \equiv \mathrm{bi}$	$\mathrm{bi} \circ \mathrm{bi} \equiv ?$	$\mathrm{bi} \circ = \equiv =$
$= \circ + \equiv =$	$= \circ - \equiv =$	$= \circ \times \equiv =$	$= \circ \mathrm{bi} \equiv =$	$= \circ = \equiv =$

Composing two bivariant functions is possibly a problem as we end up with four morphisms that are not necessarily two by two equal, and it is not obvious which one we should retain if we want to see the composed function as bivariant.

We can now try to define a bridge type as follows:

Inductive type family 3.5.1. Given a type $A : \mathcal{U}_k$, we define the type family of **bridges** $\square \frown_A \square : A \overset{\mathrm{bi}}{\to} A \overset{\mathrm{bi}}{\to} \mathcal{U}_k$ with constructors:

- $\mathsf{idbr} : \prod^{=}_{a:A} a \frown_A a$,
- $a \frown_A b$ is bivariant in a,
- $a \frown_A b$ is bivariant in b.

This definition allows us to prove properties 1-3 and 5-6. However, it also lets us compose bridges without satisfying property 7: by induction, we may assume $\beta \equiv \mathsf{id}\, a$, in which case $\beta^\dagger \circ \beta$ computes to $\mathsf{id}\, a$ so that the property is proven by reflexivity.

As pre-relations Neglecting the fact that there is more than a single universe, we could define the bridge type as follows:

$$a \frown_A b :\equiv \prod_{C:A \overset{\times}{\to} \mathcal{U}} \mathrm{Rel}(C(a), C(b)). \tag{3.92}$$

That is: a bridge from a to b is a proof that for any invariant type family C over A (and hence for *any* type family C over A), you get a relation between $C(a)$ and $C(b)$. It allows us to prove properties 1, 2, the weak version of 4 and 5-7. Property 3 however looks problematic, and even in the event where we distinguish between strongly and weakly isovariant functions, it should still be satisfied for the strongly isovariant ones.

As zigzags We could actually define $a \frown_A b$ as the type of finite zigzags from a to b. However, as might be expected, property 1 leads to contradictions for $\mathsf{strip} : A \overset{\times}{\to} A^{\mathsf{core}}$. Indeed, if we have a morphism $a \rightsquigarrow_A b$, it leads to a bridge $\mathsf{strip}\, a \frown_{A^{\mathsf{core}}} \mathsf{strip}\, b$. Now we have co- and a contravariant functions $\mathsf{unstrip} : A^{\mathsf{core}} \overset{\pm}{\to} A$ whose invariant weakenings are equal. But we can actually extract the nodes and the morphisms of our zigzag in A^{core}, and each morphism $c_i \rightsquigarrow_{A^{\mathsf{core}}} c_{i+1}$ is mapped by the covariant $\mathsf{unstrip}$ to $\mathsf{unstrip}\, c_i \rightsquigarrow_A \mathsf{unstrip}\, c_{i+1}$ and by the contravariant one to $\mathsf{unstrip}\, c_{i+1} \rightsquigarrow_A \mathsf{unstrip}\, c_i$. Then we can compose these morphisms and arrive at morphisms between a and b in either direction. Applying this to the universe, we get a function $\mathbf{1} \overset{+}{\to} \mathbf{0}$ because there is a function $\mathbf{0} \overset{+}{\to} \mathbf{1}$.

Rethinking invariance An invariant function $f : A \overset{\times}{\to} C$ that is built by using exclusively covariant and contravariant functions, can actually be modelled as $f' : A \overset{-}{\to} A \overset{+}{\to} C$, where we interpret $f(a)$ as $f'(a, a)$. We could actually try to take this as the definition of an invariant function. Then A^{core} has a single constructor

- $\mathsf{strip} : A \overset{-}{\to} A \overset{+}{\to} A^{\mathsf{core}}$,

so $A^{\mathsf{core}} \overset{+}{\simeq} A^{\mathsf{op}} \times A$. Invariant induction principles will no longer work, which is no big deal since we only ever used them in our discussion of bridged types, and the results we found there, *should* be invalidated, as the criterion $A \overset{+}{\simeq} A \times A^{\mathsf{op}}$ seems uninteresting and is not fulfilled by a type such as $\mathbb{N}$.

In particular, this means that we have a type family $\square, \square =_A \square, \square : A \overset{-}{\to} A \overset{+}{\to} A \overset{-}{\to} A \overset{+}{\to} \mathcal{U}_k$, with constructor $\mathsf{refl} : \prod^{=}_{a:A} a, a =_A a, a$. Although this is weird, there may exist an abstract interpretation for the type $a, b =_A c, d$. However, consider the type

$$\prod^{+}_{\varphi : a \rightsquigarrow b} \prod^{+}_{\chi : b \rightsquigarrow c} T(\chi \circ \varphi). \tag{3.93}$$

This type family is clearly invariant in b, which means that it actually takes a covariant b and a contravariant b. Now if these are not equal, then the term $\chi \circ \varphi$ makes no sense. A possible solution is to require that there is also a morphism from the

contravariant b to the covariant one. Then an invariant function $f : A \overset{\times}{\to} C$ is the same as a function $f' : \prod^{-}_{x:A} \prod^{+}_{y:A} (x \rightsquigarrow_A y) \overset{+}{\to} C$, interpreting $f(a)$ as $f'(a, a, \mathsf{id}\, a)$. In that case, A^{core} becomes equivalent to $\sum^{-}_{x:A} \sum^{+}_{y:A} (x \rightsquigarrow_A y)$, and at least we see that $\mathbb{N} \overset{+}{\simeq} \mathbb{N}^{\mathsf{core}}$.

Note that this means that invariant functions f map morphisms $\varphi : a \rightsquigarrow_A b$ to cospans

$$f(a, a, \mathsf{id}\, a) \rightsquigarrow f(a, b, \varphi) \leftsquigarrow f(b, b, \mathsf{id}\, b), \tag{3.94}$$

so perhaps we can think of bridges as cospans. The problem is that although covariant and isovariant functions preserve cospans, contravariant functions g map them to spans

$$g(f(a, a, \mathsf{id}\, a)) \leftsquigarrow g(f(a, b, \varphi)) \rightsquigarrow g(f(b, b, \mathsf{id}\, b)), \tag{3.95}$$

and invariant functions would map them to a double cospan (one for each branch of the original one). As not every type has pushouts, we cannot always compose cospans or turn spans into cospans, so the composition table for variances becomes problematic. On top of that, there will be problems with weakening arbitrary higher order variances to invariance.

3.5.3 An axiomatic treatment of bridges

As we will see in chapter 4, the existence of a bridge type with the desired properties doesn't seem to be outrageous. For want of a better solution, we add the bridge type to the theory using an alarmingly long series of axioms (and a few inference rules for computation in specific cases), and we will avoid its use as much as possible. The main reason to add it nonetheless, is because it is a necessary ingredient of the semantics in chapter 4. The axioms are partly based on chapter 4, partly on intuition and partly on what I want to be true. These are prototypes, especially the variance annotations may be completely off.

Axiom 3.5.2. There is a type family $\square \frown_{\square} \square : \prod^{+}_{A:\mathcal{U}_k} a \overset{\times}{\to} b \overset{\times}{\to} \mathcal{U}_k$. For every type $A :^{=} \mathcal{U}_k$, it satisfies the following properties:

- There is an identity bridge at every point: $\mathsf{idbr} : \prod^{=}_{a:A} a \frown_A a$,
- There is an anti-involution $\square^{\dagger} : \prod^{=}_{a,b:A} (a \frown_A b) \overset{+}{\simeq} (b \frown_A a)$, so that $(\mathsf{idbr}\, a)^{\dagger} \equiv \mathsf{idbr}\, a$,
- A morphism implies a bridge, invariantly: $\mathsf{toBrid} : \prod^{=}_{a,b:A} (a \rightsquigarrow_A b) \overset{\times}{\to} (a \frown_A b)$, and $\mathsf{toBrid}(\mathsf{id}\, a) \equiv \mathsf{idbr}\, a$.

In the universe, a bridge $\beta : A \frown B$ implies a cospan, i.e. a relation[a], invariantly:

$$A \xrightarrow{\mathsf{leftTransport}(\beta)} \mathsf{Rel}(\beta) \xleftarrow{\mathsf{rightTransport}(\beta)} B. \tag{3.96}$$

- $\mathsf{Rel} : \prod^{\times}_{A,B:\mathcal{U}_k} (A \frown B) \xrightarrow{+} \mathcal{U}_k$,
  so that $\mathsf{Rel}(\mathsf{idbr}\, A) \equiv A$,
  and $\mathsf{Rel}(\beta^\dagger) \equiv \mathsf{Rel}(\beta)$,
- $\mathsf{leftTransport} : \prod^{=}_{A,B:\mathcal{U}_k} \prod^{+}_{\beta:A\frown B} A \xrightarrow{+} \mathsf{Rel}(\beta)$,
  so that $\mathsf{leftTransport}(\mathsf{idbr}\, A) \equiv \mathrm{id}_A$,
  and $\mathsf{leftTransport}(\beta^\dagger) \equiv \mathsf{rightTransport}(\beta)$,
- $\mathsf{rightTransport} : \prod^{=}_{A,B:\mathcal{U}_k} \prod^{+}_{\beta:A\frown B} B \xrightarrow{+} \mathsf{Rel}(\beta)$,
  so that $\mathsf{rightTransport}(\mathsf{idbr}\, A) \equiv \mathrm{id}_A$,
  and $\mathsf{rightTransport}(\beta^\dagger) \equiv \mathsf{leftTransport}(\beta)$

For any types $A, C :^{=} \mathcal{U}_k$, and every function $f :^{=} A \xrightarrow{\times} C$, we have:

- The function f preserves bridges: $f\frown : \prod^{=}_{a,b:A} \prod^{?}_{\beta:a\frown b} f(a) \frown_C f(b)$.

So if $C : A \xrightarrow{\times} \mathcal{U}_k$ is an invariant type family and $\beta : a \frown_A b$, then we get a cospan $C\frown(\beta)$ whose components will be denoted by:

$$C(a) \xrightarrow{\beta_*} \mathsf{Rel}_C(\beta) \xleftarrow{\beta^*} C(b). \tag{3.97}$$

- For any $A :^{=} \mathcal{U}_k$ and any $C :^{=} A \xrightarrow{-} A \xrightarrow{+} \mathcal{U}_k$, define $D :\equiv a \overset{\times}{\mapsto} C(a,a)$. The above cospan for the type family D equals:

$$C(a,a) \xrightarrow{\varphi_*} C(a,b) \xleftarrow{\varphi^*} C(b,b). \tag{3.98}$$

- More generally, a path/morphism in a type family C along a morphism φ should be the same as a path/morphism between the transports to $\mathsf{Rel}_C(\mathsf{toMorph}(\varphi))$.

Now that we know that, we can state how dependent functions preserve bridges. Let $A :^{=} \mathcal{U}_k$ be a type and $C :^{=} A \xrightarrow{\times} \mathcal{U}_k$ an invariant type family. Then for any $f :^{=} \prod^{v}_{a:A} C(a)$, we have

- If $v \not\equiv =$, then f maps bridges to bridges:

$$f\frown : \prod_{a,b:A}^{=} \prod_{\beta:a\frown b}^{?} \beta_*(f(a)) \frown_{\mathsf{Rel}_C(\beta)} \beta^*(f(b)), \tag{3.99}$$

  so that

  1. $f\frown(\mathsf{idbr}\, a) \equiv \mathsf{idbr}\, f(a) : f(a) \frown_{C(a)} f(a)$,
  2. $f\frown(\beta^\dagger) \equiv f\frown(\beta)^\dagger$,
  3. If f is invariant and non-dependent, then $f\frown(\beta)$ in this sense is judgementally equal to the previous sense,

4. If f is covariant, applying f commutes with toBrid:

$$\begin{array}{ccc} \varphi : (a \rightsquigarrow b) & \xmapsto{f^{\rightsquigarrow}} & \varphi_*(f(a)) \rightsquigarrow \varphi^*(f(b)) \\ \Big\downarrow{\scriptstyle \mathsf{toBrid}} & & \Big\downarrow{\scriptstyle \mathsf{toBrid}} \\ \beta : (a \frown b) & \xmapsto[f^{\frown}]{} & \varphi_*(f(a)) \frown \varphi^*(f(b)). \end{array} \tag{3.100}$$

5. If f is contravariant, applying f commutes with toBrid up to an anti-involution:

$$\begin{array}{ccc} \varphi : (a \rightsquigarrow b) & \xmapsto{f^{\rightsquigarrow}} & \varphi^*(f(b)) \rightsquigarrow \varphi_*(f(a)) \\ & & \Big\downarrow{\scriptstyle \mathsf{toBrid}} \\ \Big\downarrow{\scriptstyle \mathsf{toBrid}} & & \varphi^*(f(b)) \frown \varphi_*(f(a)) \\ & & \Big\downarrow{\scriptstyle \square^\dagger} \\ \beta : (a \frown b) & \xmapsto[f^{\frown}]{} & \varphi_*(f(a)) \frown \varphi^*(f(b)). \end{array} \tag{3.101}$$

- If f is isovariant, then it maps bridges to paths:

$$f^{\frown} : \prod_{a,b:A}^{=} \prod_{\beta : a \frown b}^{+} \beta_*(f(a)) =_{\mathsf{Rel}_{C(\beta)}} \beta^*(f(b)), \tag{3.102}$$

so that

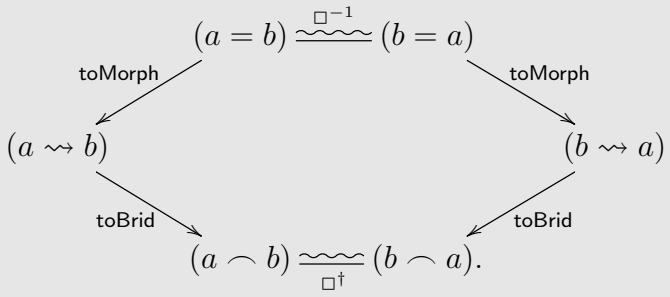
1. $f^{\frown}(\mathsf{idbr}\, a) \equiv \mathsf{refl}\, f(a) : f(a) =_{C(a)} f(a)$,
2. $f^{\frown}(\beta^\dagger) \equiv f^{\frown}(\beta)^{-1}$,
3. Applying f commutes with weakening to a bridge:

$$\begin{array}{ccc} \varphi : (a \rightsquigarrow b) & & \\ \Big\downarrow{\scriptstyle \mathsf{toBrid}} & \overset{f^{\rightsquigarrow}}{\searrow} & \\ & & \varphi_*(f(a)) = \varphi^*(f(b)) \\ & \underset{f^{\frown}}{\nearrow} & \\ \beta : (a \frown b). & & \end{array} \tag{3.103}$$

[a] The reason we choose for a cospan, rather than a span, is that a cospan $A \to X \leftarrow B$ encodes equality and morphisms between elements $a : A$ and $b : B$ in a much simpler way than a span could: we just transport both to X and compare them there.

With that, we can prove the following simple lemma:

Lemma 3.5.3. For any type $A :^{=} \mathcal{U}_k$ and any $a, b :^{=} A$, the following diagram commutes:



(3.104)

Proof. We start in $(a = b)$ and use path induction:

$$\mathsf{toBrid}(\mathsf{toMorph}(\mathsf{refl}\, a))^\dagger \equiv \mathsf{toBrid}(\mathsf{id}\, a)^\dagger \equiv (\mathsf{idbr}\, a)^\dagger \equiv \mathsf{idbr}\, a, \tag{3.105}$$

$$\mathsf{toBrid}(\mathsf{toMorph}((\mathsf{refl}\, a)^{-1})) \equiv \mathsf{toBrid}(\mathsf{toMorph}(\mathsf{refl}\, a)) \equiv \mathsf{toBrid}(\mathsf{id}\, a) \equiv \mathsf{idbr}\, a. \qquad \square$$

We will not use bridges again in the remainder of this chapter. Chapter 4 sketches a possible consistency proof which would also show consistency of the above axioms. Looking back, we see that all 7 criteria for a good bridge type are satisfied, the 7th one by not allowing composition. It is probably possible to axiomatize a composable bridge type, which would coincide with zigzags in many types, but it is more complicated. For example, we would want to be able to remove identity morphisms from zigzags and cancel out inverses.

3.6 Homotopies and natural transformations

This section is based on [Uni13, §2.4].

In symmetric HoTT

In absence of function extensionality (see section 3.8.8) and the univalence axiom (see section 3.8.9), there turns out to be some freedom in what an equality $p : f = g$ between functions $f, g : \prod_{x:A} C(x)$ means. So we should ask ourselves: what should it mean? In classical mathematics, two functions are equal if and only if they are pointwise equal. We can define an analogous notion in type theory:

$$f \sim g :\equiv \prod_{x:A} f(x) =_{C(x)} g(x). \tag{3.106}$$

We call an element of $f \sim g$ a **homotopy** or **natural equivalence** from f to g, and the following lemma explains why:

Lemma 3.6.1. Let $f, g : A \to C$ be functions and $H : f \sim g$ be a homotopy. If we

have elements $a, b : A$ and $p : a =_A b$, then the following diagram commutes:

$$\begin{array}{ccc} f(a) & \overset{H(a)}{=\!=} & g(a) \\ {\scriptstyle f^{=}(p)}\Big\| & & \Big\|{\scriptstyle g^{=}(p)} \\ f(b) & \underset{H(b)}{=\!=} & g(b). \end{array} \tag{3.107}$$

In other words, we have $\prod_{a,b:A} \prod_{p:a=b} (g^{=}(p) \circ H(a)) =_{f(a)=g(b)} (H(b) \circ f^{=}(p))$.

Proof. After path induction, we must prove $\prod_{a:A} (g^{=}(\mathsf{refl}\, a) \circ H(a)) = (H(a) \circ f^{=}(\mathsf{refl}\, a))$, and both sides compute to $H(a)$. □

In the topological interpretation of HoTT, this shows that a homotopy $H : f \sim g$ consists not only of a path $f(a) = g(a)$ for every $a : A$, but it actually fills all the squares induced by paths in A. That is, it is also a homotopy in the topological meaning of the word.

In the groupoid interpretation, this shows that a homotopy $H : f \sim g$ is actually a natural equivalence of f and g.

Lemma 3.6.2. Equal functions are homotopic:

$$\mathsf{happly} : \prod_{f,g:\prod_{x:A} C(x)} (f = g) \to (f \sim g). \tag{3.108}$$

Proof. After path induction, we can use pointwise reflexivity. □

Remark 3.6.3. The function extensionality axiom in 3.8.8 will assert that a path between functions is the *same* as a homotopy, i.e. that the above lemma is an equivalence.

In directed HoTT

In directed HoTT, we can define symmetric homotopies as well as a directed variant which we will call **natural transformations**. We only consider covariant functions $f, g : \prod^{+}_{x:A} C(x)$ as any function can be turned into a covariant one using variance currying. The only thing we have to decide is the variance on the product sign, and we will pick isovariance because we want commuting diagrams rather than awkward morphisms between both branches of the diagram:

$$f \sim g :\equiv \overset{=}{\prod_{x:A}} f(x) =_{C(x)} g(x), \tag{3.109}$$

$$f \succ g :\equiv \overset{=}{\prod_{x:A}} f(x) \rightsquigarrow_{C(x)} g(x). \tag{3.110}$$

Lemma 3.6.4. Let $f, g :^{=} A \xrightarrow{+} C$ be functions and $H :^{=} f \sim g$ be a homotopy. If we have elements $a, b :^{=} A$ and $p :^{+} a =_A b$ or $\varphi :^{+} a \rightsquigarrow_A b$, then the following diagrams

commute:

$$\begin{array}{ccc} f(a) & \overset{H(a)}{=\!=} & g(a) \\ {\scriptstyle f^{=}(p)}\Big\| & & \Big\|{\scriptstyle g^{=}(p)} \\ f(b) & \underset{H(b)}{=\!=} & g(b). \end{array} \qquad \begin{array}{ccc} f(a) & \overset{H(a)}{=\!=} & g(a) \\ {\scriptstyle f^{\rightsquigarrow}(\varphi)}\big\downarrow & & \big\downarrow{\scriptstyle g^{\rightsquigarrow}(\varphi)} \\ f(b) & \underset{H(b)}{=\!=} & g(b). \end{array} \tag{3.111}$$

In other words, we have

$$\prod_{a,b:A}^{=} \prod_{p:a=b}^{+} (g^{=}(p) \circ H(a)) =_{f(a)=g(b)} (H(b) \circ f^{=}(p)),$$

$$\prod_{a,b:A}^{=} \prod_{\varphi:a\rightsquigarrow b}^{+} (g^{\rightsquigarrow}(\varphi) \circ \mathsf{toMorph}(H(a))) =_{f(a)\rightsquigarrow g(b)} (\mathsf{toMorph}(H(b)) \circ f^{\rightsquigarrow}(\varphi)).$$

Proof. This follows immediately from the meaning of a path along a path/morphism in the type family $x \overset{\times}{\mapsto} f(x) = g(x)$, and is also straightforwardly proven using path or morphism induction. □

Lemma 3.6.5. Let $f, g :^{=} A \overset{+}{\to} C$ be functions and $N :^{=} f \succ g$ be a homotopy. If we have elements $a, b :^{=} A$ and $p :^{+} a =_A b$ or $\varphi :^{+} a \rightsquigarrow_A b$, then the following diagrams commute:

$$\begin{array}{ccc} f(a) & \overset{N(a)}{\rightsquigarrow} & g(a) \\ {\scriptstyle f^{=}(p)}\Big\| & & \Big\|{\scriptstyle g^{=}(p)} \\ f(b) & \underset{N(b)}{\rightsquigarrow} & g(b). \end{array} \qquad \begin{array}{ccc} f(a) & \overset{N(a)}{\rightsquigarrow} & g(a) \\ {\scriptstyle f^{\rightsquigarrow}(\varphi)}\big\downarrow & & \big\downarrow{\scriptstyle g^{\rightsquigarrow}(\varphi)} \\ f(b) & \underset{N(b)}{\rightsquigarrow} & g(b). \end{array} \tag{3.112}$$

In other words, we have

$$\prod_{a,b:A}^{=} \prod_{p:a=b}^{+} (\mathsf{toMorph}(g^{=}(p)) \circ N(a)) =_{f(a)\rightsquigarrow g(b)} (N(b) \circ \mathsf{toMorph}(f^{=}(p))),$$

$$\prod_{a,b:A}^{=} \prod_{\varphi:a\rightsquigarrow b}^{+} (g^{\rightsquigarrow}(\varphi) \circ N(a)) =_{f(a)\rightsquigarrow g(b)} (N(b) \circ f^{\rightsquigarrow}(\varphi)).$$

Proof. This follows immediately from the meaning of a morphism along a path/morphism in the type family $x \overset{\times}{\mapsto} f(x) = g(x)$, and is also straightforwardly proven using path or morphism induction. □

Lemma 3.6.6. For any type $A :^{=} \mathcal{U}_k$ and any type family $C :^{=} A \overset{\times}{\to} \mathcal{U}_k$, we have

that equal functions in $\prod^+_{x:A} C(x)$ are homotopic,

$$\mathsf{happly} : \prod^{=}_{f,g:\prod^+_{x:A} C(x)} (f = g) \xrightarrow{+} (f \sim g). \tag{3.113}$$

and that a morphism between functions implies a natural transformation:

$$\mathsf{napply} : \prod^{=}_{f,g:\prod^+_{x:A} C(x)} (f \rightsquigarrow g) \xrightarrow{+} (f \succ g). \tag{3.114}$$

Proof. After path/morphism induction, we can use pointwise reflexivity/identity. □

Remark 3.6.7. The function extensionality axiom in 3.8.8 will assert that a path/morphism between functions is the *same* as a homotopy/natural transformation, i.e. that the above lemma is an equivalence.

Remark 3.6.8. In particular, if we have a function $f : A \xrightarrow{+} B \xrightarrow{+} C$ and morphisms (or paths, or one morphism and one path) $\alpha : a \rightsquigarrow_A a'$ and $\beta : b \rightsquigarrow_B b'$, then the following diagram commutes:

$$\begin{array}{ccc} f(a,b) & \xrightarrow{f([\alpha],b)} & f(a',b) \\ {\scriptstyle f(a,[\beta])}\downarrow & & \downarrow{\scriptstyle f(a',[\beta])} \\ f(a,b') & \xrightarrow[f([\alpha],b')]{} & f(a',b'), \end{array} \tag{3.115}$$

where $f([\alpha], b)$ is defined as the natural transformation obtained from $f^{\rightsquigarrow}(\alpha) : f(a) \rightsquigarrow f(b)$, applied to b, and $f(a, [\beta])$ is defined as $f(a)^{\rightsquigarrow}(\beta)$. This is just a special case of the results in this section.

3.7 Groupoids

In section 2.12, we introduced bridged and bridgeless groupoids. The basic property of bridged groupoids, that +, - and = are interchangeable for functions whose domain is a bridged groupoid, was proven there. However, we have been a bit short on the nature of these equivalences, and even more so when we treated bridgeless groupoids. In this section, we analyse these equivalences properly and prove a few additional results.

Bridged groupoids

In section 2.12, we defined a bridged groupoid as a type G for which the function $\mathsf{unstrip} : G^{\mathsf{core}} \xrightarrow{+} G$ is an equivalence. We proved that for any bridged groupoid G and any type family $C : G \xrightarrow{\times} \mathcal{U}_k$, we have

$$\left(\prod_{x:G}^{\times} C(x)\right) \overset{+}{\simeq} \left(\prod_{x:G}^{+} C(x)\right) \overset{+}{\simeq} \left(\prod_{x:G}^{-} C(x)\right). \tag{3.116}$$

However, this doesn't say what these equivalences do. Fortunately, we have the following lemma:

Lemma 3.7.1. Let $G :\equiv \mathcal{U}_k$ be a bridged groupoid, and $C :\equiv G \xrightarrow{\times} \mathcal{U}_k$ a type family. Then the weakening to invariance of co- or contravariant functions from G into the type family C, is an equivalence: $\mathsf{isEquiv}(\square_\times)$, where $\square_\times$ is defined as

$$\square_\times :\equiv f \overset{+}{\mapsto} (x \overset{\times}{\mapsto} f(x)) : \left(\prod_{x:G}^{\pm} C(x)\right) \xrightarrow{+} \left(\prod_{x:G}^{\times} C(x)\right). \tag{3.117}$$

Proof. We construct an inverse s to weakening. Let $g : \prod_{x:G}^{\times} C(x)$ be invariant. Then $r(g) :\equiv \mathsf{ind}^{\pm}_{G^{\mathsf{core}}}(x \overset{\times}{\mapsto}{}' C(\mathsf{unstrip}\, x'), g)$ is a covariant or contravariant function $\prod^{\pm}_{x':G^{\mathsf{core}}} C(\mathsf{unstrip}\, x')$. To see that it is correctly typed, note that $g(x) : C(x) \equiv C(\mathsf{unstrip}\,(\mathsf{strip}\, x))$, as required by the induction principle. The result computes: $r(g)(\mathsf{strip}\, x) \equiv g(x) : C(\mathsf{unstrip}\,(\mathsf{strip}\, x))$. Now define $s(g) :\equiv r(g) \circ \mathsf{unstrip}^{-1}$.

To see that s is left inverse to weakening, take $f : \prod^{\pm}_{x:G} C(x)$. We have to show that $s(f_\times) = f$. We first show that $s(f_\times) \circ \mathsf{unstrip} = f \circ \mathsf{unstrip}$. Take $x' :^{\pm} G^{\mathsf{core}}$. By induction, we may assume that $x' \equiv \mathsf{strip}\, x$ for some $x :^{\times} G$. Then

$$\begin{aligned}(s(f_\times) \circ \mathsf{unstrip})(x') &\equiv r(f_\times)(\mathsf{unstrip}^{-1}(\mathsf{unstrip}\,(\mathsf{strip}\, x))) \\ &\equiv r(f_\times)(\mathsf{strip}\, x) \equiv f_\times(x) \equiv f(x) \equiv f(\mathsf{unstrip}\, x').\end{aligned} \tag{3.118}$$

Now we compose both functions with $\mathsf{unstrip}^{-1}$, finding

$$s(f_\times) = s(f_\times) \circ \mathsf{unstrip} \circ \mathsf{unstrip}^{-1} = f \circ \mathsf{unstrip} \circ \mathsf{unstrip}^{-1} = f. \tag{3.119}$$

To see that s is right inverse to weakening, take $g : \prod^{\times}_{x:G} C(x)$. We have to show that $s(g)_\times = g$. The reasoning is identical, as weakening doesn't really do anything. □

Now that we know that these equivalences are very well-behaved, we can denote each of these types as $\prod^{3}_{x:G} C(x)$ and implicitly applying the equivalences whenever necessary.

Lemma 3.7.2. Let $G :\equiv \mathcal{U}_k$ be a bridged groupoid. Then we have

$$\prod_{x,y:G}^{=} (x \rightsquigarrow_G y) \overset{+}{\simeq} (x =_G y). \tag{3.120}$$

Proof. ($\to$) As the type family $x \overset{3}{\mapsto} y \overset{3}{\mapsto} (x =_G y)$ has the proper variance, we can apply morphism induction and map $x, x, \mathsf{id}\, x$ to $\mathsf{refl}\, x$.

($\leftarrow$) By path induction, we can map $x, x, \mathsf{refl}\, x$ to $\mathsf{id}\, x$.
These functions are clearly inverses. □

Bridgeless groupoids

In section 2.12, we defined a bridgeless groupoid as a type G with a function $i : G \overset{=}{\to} G$ that weakens to identity: $i_+ = \mathrm{id}_G$.

Lemma 3.7.3. Every bridgeless groupoid is a bridged groupoid:

$$\prod_{G:\mathcal{U}_k}^{=} \mathsf{isBridgelessGrpd}(G) \overset{+}{\to} \mathsf{isBridgedGrpd}(G). \tag{3.121}$$

Proof. Let $i : G \overset{=}{\to} G$ be the isovariant identity function. Then $(\mathsf{strip} \circ i)_+$ is easily shown to be an inverse of $\mathsf{unstrip}$. □

We argued earlier that for a bridgeless groupoid G,

$$\left(\prod_{x:G}^{\times} C(x)\right) \overset{+}{\simeq} \left(\prod_{x:G}^{+} C(x)\right) \overset{+}{\simeq} \left(\prod_{x:G}^{-} C(x)\right) \overset{+}{\simeq} \left(\prod_{x:G}^{=} C(x)\right). \tag{3.122}$$

The following lemma characterizes the equivalences between isovariant functions on one hand and co- or contravariant functions on the other hand:

Lemma 3.7.4. Let $G :\equiv \mathcal{U}_k$ be a bridgeless groupoid, and $C :\equiv G \overset{\times}{\to} \mathcal{U}_k$ a type family. Then the weakening to co- or contravariance of isovariant functions from G into the type family C, is an equivalence: $\mathsf{isEquiv}(\square_\pm)$, where $\square_\pm$ is defined as

$$\square_\pm :\equiv f \overset{+}{\mapsto} (x \overset{\pm}{\mapsto} f(x)) : \left(\prod_{x:G}^{=} C(x)\right) \overset{+}{\to} \left(\prod_{x:G}^{\pm} C(x)\right). \tag{3.123}$$

Proof. It is easy to show that $f \overset{+}{\mapsto} f \circ i$ is an inverse to either weakening. □

Thus, we can denote all four function types as $\prod^4_{x:G} C(x)$, implicitly applying equivalences where necessary.

If the codomain is a bridgeless groupoid, variance doesn't matter either. We can even show this for dependent functions:

Lemma 3.7.5. Let $A :\equiv \mathcal{U}_k$ be a type, $G :\equiv A \overset{\times}{\to} \mathcal{U}_k$ a type family and $p : \prod^{=}_{a:A} \mathsf{isBridgelessGrpd}(G(a))$ a proof that G is a family of bridgeless groupoids in

a natural way. Then for any variance v, the weakening function

$$\square_v : \left(\prod_{a:A}^{=} G(a)\right) \xrightarrow{+} \left(\prod_{a:A}^{v} G(a)\right) \tag{3.124}$$

is an equivalence.

Proof. Let i_a be the isovariant identity for $G(a)$. Note that it depends isovariantly on a because p is isovariant. We define an inverse s for $\square_v$, by setting $s(f) :\equiv a \overset{=}{\mapsto} i_a(f(a))$. Now $s(f)(a) \equiv i_a(f(a)) = \mathrm{id}_{G(a)}(f(a)) \equiv f(a)$. Since weakening doesn't do anything, this allows one to prove that s is an inverse for $\square_v$. □

Thus, we get

$$\left(\prod_{a:A}^{4} G(a)\right) :\overset{+}{\simeq} \left(\prod_{a:A}^{\times} G(a)\right) \overset{+}{\simeq} \left(\prod_{a:A}^{+} G(a)\right) \overset{+}{\simeq} \left(\prod_{a:A}^{-} G(a)\right) \overset{+}{\simeq} \left(\prod_{a:A}^{=} G(a)\right). \tag{3.125}$$

3.8 The higher category structure of some basic types

The definitions of the identity and morphism types are rather mysterious. In this section, we investigate what the meaning of equality and morphisms actually is. Often, we will be able to prove that they mean what one would expect they mean. In a few cases, we will only be able to prove that they *imply* what one would expect they mean; the other implication will have to be assumed by axiom. The characterizations of the identity type generally also hold in the symmetric case and can be obtained by removing variance annotations.

3.8.1 The coproduct

This section is based on [Uni13, §2.12]. We define a type family $P : A+B \overset{\times}{\to} A+B \overset{\times}{\to} \mathcal{U}_k$ by induction on the coproduct as follows:

$$\begin{aligned} P(\mathsf{inl}\, a, \mathsf{inl}\, a') &:\equiv a =_A a', \\ P(\mathsf{inl}\, a, \mathsf{inr}\, b') &:\equiv \mathbf{0}, \\ P(\mathsf{inr}\, b, \mathsf{inl}\, a') &:\equiv \mathbf{0}, \\ P(\mathsf{inr}\, b, \mathsf{inr}\, b') &:\equiv b =_B b'. \end{aligned} \tag{3.126}$$

Lemma 3.8.1. For any two types $A, B :^{=} \mathcal{U}_k$ and for all $x, y :^{=} A + B$, the type of paths from x to y is equivalent to $P(x, y)$ (as defined above):

$$\prod_{x,y:A+B}^{=} (x = y) \overset{+}{\simeq} P(x, y). \tag{3.127}$$

This basically means that elements of $A + B$ are equal if and only if they come from the same side and they were equal there.

Proof. ($\to$) After path induction, we have to prove

$$\prod_{x:A+B}^{=} C(x, x). \tag{3.128}$$

After induction on x, we only need

$$\prod_{a:A}^{=} P(\mathsf{inl}\, a, \mathsf{inl}\, a), \qquad \prod_{b:B}^{=} P(\mathsf{inr}\, b, \mathsf{inr}\, b). \tag{3.129}$$

Since $P(\mathsf{inl}\, a, \mathsf{inl}\, a) \equiv (a =_A a)$ and $P(\mathsf{inr}\, b, \mathsf{inr}\, b) \equiv (b =_B b)$, we can prove this by reflexivity. So we are mapping $\mathsf{inl}\, a, \mathsf{inl}\, a, \mathsf{refl}(\mathsf{inl}\, a)$ to $\mathsf{refl}\, a$ and $\mathsf{inr}\, b, \mathsf{inr}\, b, \mathsf{refl}(\mathsf{inr}\, b)$ to $\mathsf{refl}\, b$.

($\leftarrow$) We apply induction to x and y, leading to four cases. When x and y come from different sides, we have to prove $\mathbf{0} \xrightarrow{+} (x = y)$ which is automatically true by induction on the empty type. In the other two cases, we can subsequently apply path induction and use $\mathsf{refl}(\mathsf{inl}\, a)$ and $\mathsf{refl}(\mathsf{inr}\, b)$. So we are mapping $a, a, \mathsf{refl}\, a$ to $\mathsf{refl}(\mathsf{inl}\, a)$ and $b, b, \mathsf{refl}\, b$ to $\mathsf{refl}(\mathsf{inr}\, b)$, which is clearly the inverse of the other arrow. □

We also define a type family $M : A + B \xrightarrow{\times} A + B \xrightarrow{\times} \mathcal{U}_k$ by

$$\begin{aligned} M(\mathsf{inl}\, a, \mathsf{inl}\, a') &:\equiv a \rightsquigarrow_A a', \\ M(\mathsf{inl}\, a, \mathsf{inr}\, b') &:\equiv \mathbf{0}, \\ M(\mathsf{inr}\, b, \mathsf{inl}\, a') &:\equiv \mathbf{0}, \\ M(\mathsf{inr}\, b, \mathsf{inr}\, b') &:\equiv b \rightsquigarrow_B b'. \end{aligned} \tag{3.130}$$

Lemma 3.8.2. For any two types $A, B :^{=} \mathcal{U}_k$ and for all $x, y :^{=} A + B$, the type of morphisms from x to y is equivalent to $M(x, y)$ (as defined above):

$$\prod_{x,y:A+B}^{=} (x \rightsquigarrow y) \overset{+}{\simeq} M(x, y). \tag{3.131}$$

Proof. Analogous – the variances are always right for applying morphism induction. □

3.8.2 The naturals

This section is based on [Uni13, §2.13] We define a type family $P : \mathbb{N} \xrightarrow{4} \mathbb{N} \xrightarrow{4} \mathcal{U}_k$ by

induction on the naturals as follows:

$$\begin{aligned} P(\mathsf{zero}, \mathsf{zero}) &:\equiv \mathbf{1}, \\ P(\mathsf{zero}, \mathsf{succ}\, n) &:\equiv \mathbf{0}, \\ P(\mathsf{succ}\, m, \mathsf{zero}) &:\equiv \mathbf{0}, \\ P(\mathsf{succ}\, m, \mathsf{succ}\, n) &:\equiv P(m, n). \end{aligned} \tag{3.132}$$

Lemma 3.8.3. For all $m, n :^4 \mathbb{N}$, the type of paths from m to n is equivalent to $P(x, y)$ (as defined above):

$$\prod_{m,n:\mathbb{N}}^{=} (m = n) \overset{+}{\simeq} P(m, n). \tag{3.133}$$

This is a recursive way to say that natural numbers are only equal to themselves and only in one way.

Proof. ($\to$) By path induction, we need only prove

$$k : \prod_{n:\mathbb{N}}^{=} P(n, n). \tag{3.134}$$

We define $k(\mathsf{zero}) :\equiv \star$ and $k(\mathsf{succ}\, n) :\equiv k(n)$. So we get a function f that computes: $f_{\mathsf{zero},\mathsf{zero}}(\mathsf{refl}\, \mathsf{zero}) \equiv \star$ and $f_{\mathsf{succ}\, n,\mathsf{succ}\, n}(\mathsf{refl}(\mathsf{succ}\, n)) \equiv f_{n,n}(\mathsf{refl}\, n)$.

($\leftarrow$) We construct an inverse g. By induction on m and n, we get four cases, two of which are resolved by applying induction on $\mathbf{0}$. We set $g_{\mathsf{zero},\mathsf{zero}}(\star) :\equiv \mathsf{refl}\, \mathsf{zero}$ and $g_{\mathsf{succ}\, m,\mathsf{succ}\, n}(p) :\equiv \mathsf{succ}^{=}(g_{m,n}(p))$.

We show that $\prod^4_{m,n:\mathbb{N}} g_{m,n} \circ f_{m,n} = \mathrm{id}_{m=n}$. By function extensionality, that becomes $\prod^4_{m,n:\mathbb{N}} \prod^+_{p:m=n} g_{m,n}(f_{m,n}(p)) = p$. By path induction, we need $\eta : \prod^4_{n:\mathbb{N}} g_{n,n}(f_{n,n}(\mathsf{refl}\, n)) = \mathsf{refl}\, n$. For $n \equiv \mathsf{zero}$, both sides compute to $\mathsf{refl}\, \mathsf{zero}$ so we can take reflexivity. For $n \equiv \mathsf{succ}\, m$, the left side computes to $\mathsf{succ}^{=}(g_{m,m}(f_{m,m}(\mathsf{refl}\, m))$, but applying $\eta_{m,m}(\mathsf{refl}\, m)$ under $\mathsf{succ}^{=}$ shows that this is equal to $\mathsf{succ}^{=}(\mathsf{refl}\, m)$, which computes to $\mathsf{refl}(\mathsf{succ}\, m)$, just like the right hand side.

Conversely, we show that $\prod^4_{m,n:\mathbb{N}} f_{m,n} \circ g_{m,n} = \mathrm{id}_{P(m,n)}$. Again, function extensionality we need $\zeta : \prod^4_{m,n:\mathbb{N}} \prod^+_{p:P(m,n)} f_{m,n}(g_{m,n}(p)) = p$. By induction on m and n, we get four cases of which two are trivial. If both m and n are zero, then we can assume that $p \equiv \star$ in which case the left hand side also computes to $\star$. If both are successors, we can compute:

$$f_{\mathsf{succ}\, i,\mathsf{succ}\, j}(g_{\mathsf{succ}\, i,\mathsf{succ}\, j}(p)) \equiv f_{\mathsf{succ}\, i,\mathsf{succ}\, j}(\mathsf{succ}^{=}(g_{i,j}(p))). \tag{3.135}$$

Now, since $f_{\mathsf{succ}\, i,\mathsf{succ}\, i}(\mathsf{succ}^{=}(\mathsf{refl}\, i)) \equiv f_{\mathsf{succ}\, i,\mathsf{succ}\, i}(\mathsf{refl}(\mathsf{succ}\, i)) \equiv f_{i,i}(\mathsf{refl}\, i)$, it follows by path induction that $f_{\mathsf{succ}\, i,\mathsf{succ}\, j}(\mathsf{succ}^{=}(q)) = f_{i,j}(q)$ for any path q. So we can proceed:

$$f_{\mathsf{succ}\, i,\mathsf{succ}\, j}(\mathsf{succ}^{=}(g_{i,j}(p))) = f_{i,j}(g_{i,j}(p)), \tag{3.136}$$

which is equal to p by $\zeta_{i,j}(p)$. □

Since the naturals form a bridgeless groupoid, a morphism between naturals is the same as a path, by lemma 3.7.2.

3.8.3 The opposite

Lemma 3.8.4. For any type $A :^{=} \mathcal{U}_k$, we have

$$\prod_{a',b':A^{\mathsf{op}}}^{=} (a' =_{A^{\mathsf{op}}} b') \overset{+}{\simeq} (\mathsf{unflip}\, a' =_A \mathsf{unflip}\, b'). \tag{3.137}$$

Proof. ($\to$) By path induction, we need only prove $\prod^{=}_{a':A^{\mathsf{op}}} \mathsf{unflip}\, a' =_A \mathsf{unflip}\, a'$. This could be done by reflexivity, but to get a more comprehensible function, we apply induction on a' and have to prove $\prod^{=}_{a:A} a =_A a$ as $\mathsf{unflip}(\mathsf{flip}\, a)$ computes to a. This is proven by reflexivity. So we are mapping $\mathsf{flip}\, a$, $\mathsf{flip}\, a$, $\mathsf{refl}(\mathsf{flip}\, a)$ to $\mathsf{refl}\, a$.

($\leftarrow$) By induction on a' and b', we have to prove

$$\prod_{a,b:A}^{=} (a =_A b) = (\mathsf{flip}\, a =_{A^{\mathsf{op}}} \mathsf{flip}\, b). \tag{3.138}$$

This is proven by $\mathsf{flip}^{=}$, which computes $\mathsf{flip}^{=}_{a,a}(\mathsf{refl}\, a) \equiv \mathsf{refl}(\mathsf{flip}\, a)$, so ultimately we are mapping $\mathsf{flip}\, a, \mathsf{flip}\, a, \mathsf{refl}\, a$ to $\mathsf{refl}(\mathsf{flip}\, a)$, which is clearly the inverse of the other arrow. □

Lemma 3.8.5. For any type $A :^{=} \mathcal{U}_k$, we have (note the swapping of a' and b'):

$$\prod_{a',b':A^{\mathsf{op}}}^{=} (a' \rightsquigarrow_{A^{\mathsf{op}}} b') \overset{+}{\simeq} (\mathsf{unflip}\, b' \rightsquigarrow_A \mathsf{unflip}\, a'). \tag{3.139}$$

Proof. ($\to$) By morphism induction, we need to prove $\prod^{=}_{a':A^{\mathsf{op}}} \mathsf{unflip}\, a' \rightsquigarrow_A \mathsf{unflip}\, a'$. Again, we choose to apply induction on a' and have to prove $\prod^{=}_{a:A} a \rightsquigarrow_A a$ which is proven by $\mathsf{id}\, a$. So we are mapping $\mathsf{flip}\, a$, $\mathsf{flip}\, a$, $\mathsf{id}(\mathsf{flip}\, a)$ to $\mathsf{id}\, a$

($\leftarrow$) By induction on a' and b', we have to prove

$$\prod_{a,b:A}^{=} (a \rightsquigarrow_A b) = (\mathsf{flip}\, b \rightsquigarrow_{A^{\mathsf{op}}} \mathsf{flip}\, a). \tag{3.140}$$

This is proven by $\mathsf{flip}^{\rightsquigarrow}$, which computes $\mathsf{flip}^{\rightsquigarrow}_{a,a}(\mathsf{id}\, a) \equiv \mathsf{id}(\mathsf{flip}\, a)$, so ultimately we are mapping $\mathsf{flip}\, a, \mathsf{flip}\, a, \mathsf{id}\, a$ to $\mathsf{id}(\mathsf{flip}\, a)$, which is clearly the inverse of the other arrow. □

3.8.4 The core

Lemma 3.8.6. For any type $A :^{=} \mathcal{U}_k$, we have

$$\prod_{a',b':A^{\mathsf{core}}}^{=} (a' =_{A^{\mathsf{op}}} b') \overset{+}{\simeq} (\mathsf{unstrip}\, a' =_A \mathsf{unstrip}\, b'). \tag{3.141}$$

Proof. Replace flip and unflip with strip and $\mathsf{unstrip}$ in the proof of lemma 3.8.4. □

Since a type's core is a bridged groupoid, the morphism types are equivalent to the equality types.

3.8.5 The localization

Theorem 3.8.7. For any type $A :^{=} \mathcal{U}_k$, we have the following property:

$$\prod_{a,b:A}^{=} (\mathsf{gather}\, a =_{A^{\mathsf{loc}}} \mathsf{gather}\, b) \overset{+}{\simeq} \left(\prod_{X:\mathcal{U}_k}^{=} \prod_{f:A \overset{=}{\to} X}^{=} f(a) =_X f(b) \right), \tag{3.142}$$

i.e. a proof that $\mathsf{gather}\, a = \mathsf{gather}\, b$ is the same as a proof that a and b are mapped to equal values by *any* isovariant function whose codomain is also in $\mathcal{U}_k$.

Proof. The idea of the proof is that since $(A^{\mathsf{loc}}, \mathsf{gather})$ is the initial object of the category of all tuples of a type X with an isovariant function $f : A \overset{=}{\to} X$, to say that $\mathsf{gather}\, a = \mathsf{gather}\, b$ is the same as to say that $f(a) = f(b)$ for all such tuples (X, f) in a natural way.

Pick $a, b :^{=} A$.

($\to$) Take $X :^{=} \mathcal{U}_k$ and $f :^{=} A \overset{=}{\to} X$. Then $r(f) :\equiv \mathsf{rec}^{=}_{A^{\mathsf{loc}}}(X, f)$ is a covariant function $A^{\mathsf{loc}} \overset{+}{\to} X$ that computes: $r(f)(\mathsf{gather}\, a) \equiv f(a)$. Note that $r(f)$ is isovariant in A and X and covariant in f. Now we map $p : \mathsf{gather}\, a = \mathsf{gather}\, b$ to $r(f)^{=}(p) : f(a) = f(b)$, which is isovariant in f as $\square^{=}$ is isovariant.

($\leftarrow$) Take a proof η of the right hand side. Then $\eta(A^{\mathsf{loc}}, \mathsf{gather})$ proves $\mathsf{gather}\, a = \mathsf{gather}\, b$.

Pick $a, b :^{=} A$. Call the function to the right R and the one to the left L. We have $R(p)(X, f) \equiv r(f)^{=}(p)$ and $L(\eta) \equiv \eta(A^{\mathsf{loc}}, \mathsf{gather})$.

- We show that $L \circ R = \mathrm{id}$.

$$(L \circ R)(p) \equiv L(X f \overset{=}{\mapsto} r \overset{=}{\mapsto} (f)^{=}(p)) \equiv r(\mathsf{gather})^{=}(p). \tag{3.143}$$

We now show that

$$\prod_{a',b':A^{\mathsf{loc}}}^{=} \prod_{p:a'=b'}^{+} r(\mathsf{gather})^{=}(p) = p. \tag{3.144}$$

By path induction, we need only prove $\prod_{a'}^{=} r(\mathsf{gather})^{=}(\mathsf{refl}\, a') = \mathsf{refl}\, a'$. But the left hand side computes:

$$r(\mathsf{gather})^{=}(\mathsf{refl}\, a') \equiv \mathsf{refl}(r(\mathsf{gather})(a')). \tag{3.145}$$

By induction on a', we may assume that $a' \equiv \mathsf{gather}\, c$, and then we have $r(\mathsf{gather})(a') \equiv r(\mathsf{gather})(\mathsf{gather}\, c) \equiv \mathsf{gather}\, c \equiv a'$. So we conclude that $\mathsf{refl}(r(\mathsf{gather})(a')) = \mathsf{refl}\, a'$, which proves (3.144). Then that equation applies in particular to $a' :\equiv \mathsf{gather}\, a$ and $b' :\equiv \mathsf{gather}\, b$. Then $(L \circ R)(p) = \mathrm{id}(p)$, and by function extensionality, $L \circ R = \mathrm{id}$.

- We also show that $R \circ L = \mathrm{id}$. By function extensionality, we should prove that $(R \circ L)(\eta)(X, f) = \eta(X, f)$ for all $X :^{=} \mathcal{U}_k$ and $f :^{=} A \xrightarrow{=} X$. We have

  $$(R \circ L)(\eta)(X, f) \equiv R(\eta(A^{\mathsf{loc}}, \mathsf{gather}))(X, f) \equiv r(f)^{=}(\eta(A^{\mathsf{loc}}, \mathsf{gather})) : f(a) =_X f(b).$$

  Now, $r(f)$ is a covariant function $A^{\mathsf{loc}} \xrightarrow{+} X$, so by the directed univalence axiom (see section 3.8.9) we get a morphism $\mathsf{dua}(r(f)) : A^{\mathsf{loc}} \rightsquigarrow_{\mathcal{U}_k} X$. Then $\eta^{\rightsquigarrow}$ maps $\mathsf{dua}(r(f))$ to a proof that $\eta(A^{\mathsf{loc}})$ is $\eta(X)$ along $\mathsf{dua}(r(f))$ in the type family $Z \xrightarrow{\times} \left(\prod_{f:A\xrightarrow{=}X}^{=} f(a) =_Z f(b)\right)$. One can prove that this means that for any function $h : A \xrightarrow{=} A^{\mathsf{loc}}$:

  $$r(f)^{=}(\eta(A^{\mathsf{loc}}, h)) = \eta(X, r(f) \circ h). \tag{3.146}$$

  Applying this to gather, and using that $r(f) \circ \mathsf{gather} = f$ from the computation rule of $r(f)$, we obtain

  $$r(f)^{=}(\eta(A^{\mathsf{loc}}, \mathsf{gather})) = \eta(X, f), \tag{3.147}$$

  as we needed to prove. □

Since a type's localization is a bridgeless groupoid, the morphism types in the localization are equivalent to the equality types.

The above characterization may create the impression that we should define $a \frown_A b :\equiv \mathsf{gather}\, a =_{A^{\mathsf{loc}}} \mathsf{gather}\, b$. Although this definition satisfies properties 1-3 and 5-7 from section 3.5, it violates the requirement that a bridge between types should be the same as a relation. Indeed, for any type $A : \mathcal{U}_k$, we have a function $f : A \xrightarrow{+} \mathbf{1}$. Then the directed univalence axiom gives a morphism $\mathsf{dua}(f) : A \rightsquigarrow \mathbf{1}$. But since $X \xmapsto{=} (\mathsf{gather}\, X = \mathsf{gather}\, B)$ is isovariant, we can conclude

$$(\mathsf{gather}\, A = \mathsf{gather}\, B) \overset{+}{\simeq} (\mathsf{gather}\, \mathbf{1} = \mathsf{gather}\, B). \tag{3.148}$$

However, we clearly do not have for all $A, B : \mathcal{U}_k$:

$$\mathrm{Rel}(A, B) \overset{+}{\simeq} \mathrm{Rel}(\mathbf{1}, B). \tag{3.149}$$

3.8.6 The product

This section is based on [Uni13, §2.6].

Lemma 3.8.8. For any types $A, B :^{=} \mathcal{U}_k$, we have

$$\prod_{c,c':A\times B}^{=} (c =_{A\times B} c') \overset{+}{\simeq} (\mathsf{prl}\, c =_A \mathsf{prl}\, c') \times (\mathsf{prr}\, c =_B \mathsf{prr} c'). \tag{3.150}$$

Proof.

($\to$) We can map $r : c = c'$ to $(\mathsf{prl}^{=}(c), \mathsf{prr}^{=}(c))$. The computation rule is that we map $c, c, \mathsf{refl}\, c$ to $(\mathsf{refl}(\mathsf{prl}\, c), \mathsf{refl}(\mathsf{prr}\, c))$.

($\leftarrow$) After currying, we have to prove

$$\prod_{a,a':A}^{=} \prod_{b,b':B}^{=} (a =_A a) \overset{+}{\to} (b =_B b) \overset{+}{\to} ((a, b) = (a', b')). \tag{3.151}$$

Double path induction leaves only $\prod^{=}_{a:A} \prod^{=}_{b:B} (a, b) = (a, b)$, which is proven by reflexivity. The computation rule is that we map $(a, b), (a, b), (\mathsf{refl}\, a, \mathsf{refl}\, b)$ to $\mathsf{refl}\, (a, b)$. This is clearly the inverse of the other arrow. □

Lemma 3.8.9. For any types $A, B :^{=} \mathcal{U}_k$, we have

$$\prod_{c,c':A\times B}^{=} (c \rightsquigarrow_{A\times B} c') \overset{+}{\simeq} (\mathsf{prl}\, c \rightsquigarrow_A \mathsf{prl}\, c') \times (\mathsf{prr}\, c \rightsquigarrow_B \mathsf{prr} c'). \tag{3.152}$$

Proof. Analogous. □

3.8.7 Dependent pair types

This section is based on [Uni13, §2.7]. We consider only covariant and isovariant dependent pair types, as the others can be reduced to covariant ones using the opposite and the core.

Covariant dependent pair types

Lemma 3.8.10. For any type $A :^{=} \mathcal{U}_k$ and any type family $B :^{=} A \overset{\times}{\to} \mathcal{U}_k$, we have

$$\prod_{c,c':\sum_{x:A} B(x)}^{=} (c = c') \overset{+}{\simeq} \left(\sum_{p:\mathsf{prl}\, c =_A \mathsf{prl}\, c'}^{+} \frac{\mathsf{prr}\, c =_B \mathsf{prr}\, c'}{p} \right). \tag{3.153}$$

Proof. ($\to$) We can map $c, c, \mathsf{refl}\, c$ to $\left(\mathsf{refl}(\mathsf{prl}\, c) \dagger \frac{\mathsf{refl}(\mathsf{prr}\, c)}{\mathsf{refl}(\mathsf{prl}\, c)}\right)$.

($\leftarrow$) We can map $(a \dagger b), (a \dagger b), \left(\mathsf{refl}\, a \dagger \frac{\mathsf{refl}\, b}{\mathsf{refl}\, a}\right)$ to $\mathsf{refl}\ (a \dagger b)$. This is clearly the inverse of the other arrow. □

Lemma 3.8.11. For any type $A :^{=} \mathcal{U}_k$ and any type family $B :^{=} A \overset{\times}{\to} \mathcal{U}_k$, we have

$$\prod_{c,c':\sum_{x:A} B(x)}^{=} (c \rightsquigarrow c') \overset{+}{\simeq} \left(\sum_{\varphi:\mathsf{prl}\, c \rightsquigarrow_A \mathsf{prl}\, c'}^{+} \frac{\mathsf{prr}\, c \rightsquigarrow_B \mathsf{prr}\, c'}{\varphi} \right). \tag{3.154}$$

Proof. ($\to$) We can map $c, c, \mathsf{id}\, c$ to $\left(\mathsf{id}(\mathsf{prl}\, c) \dagger \frac{\mathsf{id}(\mathsf{prr}\, c)}{\mathsf{id}(\mathsf{prl}\, c)}\right)$.

($\leftarrow$) We can map $(a \dagger b), (a \dagger b), \left(\mathsf{id}\, a \dagger \frac{\mathsf{id}\, b}{\mathsf{id}\, a}\right)$ to $\mathsf{id}\ (a \dagger b)$. This is clearly the inverse of the other arrow. □

Isovariant dependent pair types are injective limits

The fact that $(\sum_{a:A}^{=} \mathbf{1}) \overset{+}{\simeq} A^{\mathsf{loc}}$, indicates that it will be less easy to characterize equalities and morphisms in $\sum_{a:A}^{=} B(a)$. We can aim for a universal property as we had for the equation in A^{loc}:

$$\prod_{a,a':A}^{=} \prod_{b,b':B(a)}^{=} ((a \dagger b) = (a' \dagger b')) \overset{+}{\simeq} \left(\prod_{X:\mathcal{U}_k}^{=} \prod_{t:A}^{=} B(t) \overset{+}{\to} X \right). \tag{3.155}$$

The proof of the characterization of equality for A^{loc}, was based on the idea that $(A^{\mathsf{loc}}, \mathsf{gather})$ is an initial object of all pairs consisting of a type X and an isovariant function $A \overset{=}{\to} X$. In fact, the pair $(\sum_{a:A}^{=} B(a), (\square \dagger \square))$ is an initial object of all pairs consisting of a type X and a function $f : \prod_{t:A}^{=} B(t) \overset{+}{\to} X$. Now, we can also regard f as a family of functions $f_t : B(t) \overset{+}{\to} A$ indexed over A. If the type family B is covariant, the fact that this family is isovariant in t means that for every morphism $\varphi : a \rightsquigarrow_A a'$, the following diagram commutes:

$$\begin{array}{ccc} & X & \\ {\scriptstyle f_a}\nearrow & & \nwarrow{\scriptstyle f_{a'}} \\ B(a) & \xrightarrow[\varphi_*]{} & B(a'). \end{array} \tag{3.156}$$

Now if $\sum^{=}_{a:A} B(a)$ with the family of functions $(t \mathbin{⨾} \square) : B(t) \xrightarrow{+} \sum^{=}_{a:A} B(a)$ is the initial object of such constructions, that just means it is the injective limit: $\sum^{=}_{a:A} B(a) = \varinjlim_{a:A} B(a)$, with $(a \mathbin{⨾} \square)$ the natural injections.

The universal property of the injective limit $L = \varinjlim_{a:A} B(a)$ states that for any type X, functions $f : L \xrightarrow{+} X$ are in bijection with families of functions $f' : \prod^{=}_{a:A} B(a) \xrightarrow{+} X$; where the family is obtained from f by composing with the natural injections $q_a : B(a) \xrightarrow{+} L$. It automatically follows that this correspondence is natural in X, in the sense that for any function $g : X \xrightarrow{+} Y$, the family of functions corresponding to $g \circ f$ is the family of functions corresponding to f, composed with g. In directed HoTT, this translates to $\prod^{=}_{X:\mathcal{U}_k} \mathsf{isEquiv}(\kappa(X))$, where

$$\kappa :\equiv X \overset{=}{\mapsto} f \overset{=}{\mapsto} \left(a \overset{=}{\mapsto} b \overset{+}{\mapsto} (f \circ q_a)(b)\right) : \prod_{X:\mathcal{U}_k}^{=} \left(L \xrightarrow{+} X\right) \xrightarrow{+} \left(\prod_{a:A}^{=} B(a) \xrightarrow{+} X\right). \tag{3.157}$$

We know that $\kappa(X)$ is isovariant in X; the fact that the proof of $\mathsf{isEquiv}(\kappa(X))$ is isovariant in X means that the left and right inverses are also isovariant in X (and imposes well-behavedness restrictions on the proofs that they are left and right inverses).

We have the following lemma, which, for covariant B, is to the statement that $\sum^{=}_{a:A} B(a) = \varinjlim_{a:A} B(a)$.

Lemma 3.8.12. For any type $A :^{=} \mathcal{U}_k$ and any type family $B :^{=} A \xrightarrow{\times} \mathcal{U}_k$, we have $\prod^{=}_{X:\mathcal{U}_k} \mathsf{isEquiv}(\kappa(X))$ where

$$\kappa :\equiv X \overset{=}{\mapsto} f \overset{+}{\mapsto} \left(a \overset{=}{\mapsto} b \overset{+}{\mapsto} f((a \mathbin{⨾} b))\right) : \prod_{X:\mathcal{U}_k}^{=} \left(\left(\sum_{a:A}^{=} B(a)\right) \xrightarrow{+} X\right) \xrightarrow{+} \left(\prod_{a:A}^{=} B(a) \xrightarrow{+} X\right). \tag{3.158}$$

Proof. This is a special case of currying, see Remark 2.13.6 on page 55. □

We can expect to be able to prove the following:

Conjecture 3.8.13. For any type $A :^{=} \mathcal{U}_k$ and any type family $B :^{=} A \xrightarrow{\times} \mathcal{U}_k$, we have the following property:

$$\prod_{a,a':A}^{=} \prod_{b:B(a)}^{=} \prod_{b':B(a')}^{=} ((a \mathbin{⨾} b) = (a' \mathbin{⨾} b')) \overset{+}{\simeq} \left(\prod_{X:\mathcal{U}_k}^{=} \prod_{f:\prod^{=}_{t:A} B(t) \xrightarrow{+} X} f(a,b) = f(a',b')\right),$$

and the same holds when we replace $=$ with $\rightsquigarrow$.

However, the fact that $\sum^{=}_{a:A} B(a)$ is likely more interesting than a characterization of its identity and morphism types.

3.8.8 Dependent function types

The parts of this section that treat function extensionality are based on [Uni13, §2.9].

Function extensionality in symmetric HoTT

Lemma 3.6.2 in section 3.6 stated that equal functions are homotopic:[2]

$$\mathsf{happly} : \prod_{f,g:\prod_{a:A} C(a)} (f = g) \to (f \sim g). \tag{3.159}$$

We would like this to be an equivalence, so that we could characterize paths in the Π-type as homotopies. Unfortunately, classical MLTT is insufficient to prove this. Therefore, we assume

Axiom 3.8.14 (Function extensionality)**.** The function happly is an equivalence: $\mathsf{funExt} : \mathsf{isEquiv}(\mathsf{happly})$. Hence, for all $f, g : \prod_{a:A} C(a)$, we have $(f = g) \simeq (f \sim g)$.

An axiom in type theory is a bit different from an axiom in first order logic. In first order logic, stating an axiom simply means that you assume that proposition to be true. In type theory, we say that a proposition is true by giving an element of it. So here, we are assuming that $\mathsf{isEquiv}(\mathsf{happly})$ contains an element funExt, and this element is completely unknown in the sense that we have no computation rules for it.

The inverse $\mathsf{happly}^{-1} : (f \sim g) \to (f = g)$ will also be denoted funExt. We do know something about this funExt, namely that $\mathsf{funExt} \circ \mathsf{happly} = \mathrm{id}$ and $\mathsf{happly} \circ \mathsf{funExt} = \mathrm{id}$.

In homotopy type theory, the function extensionality axiom can actually be proven from the univalence axiom (see [Uni13, §4.9]) or from the existence of an interval type (a type containing two distinct points and a path between them; essentially the type theoretic equivalent of the topological space $[0, 1]$).

Function extensionality in directed HoTT

Lemma 3.6.6 in section 3.6 stated that equal functions are homotopic and a morphism between functions implies a natural transformation:

$$\mathsf{happly} : \prod^{=}_{f,g:\prod_{a:A} C(a)} (f = g) \xrightarrow{+} (f \sim g), \tag{3.160}$$

$$\mathsf{napply} : \prod^{=}_{f,g:\prod_{a:A} C(a)} (f \rightsquigarrow g) \xrightarrow{+} (f \succ g). \tag{3.161}$$

We assume:

[2] The name happly comes from [Uni13]. I think the 'h' stands for 'homotopy'.

Axiom 3.8.15 (Directed function extensionality)**.** The function napply is an equivalence: dirFunExt : isEquiv(napply). Hence, for all $f, g :^{=} \prod^{+}_{a:A} C(a)$, we have $(f \rightsquigarrow g) \simeq (f \succ g)$.

The directed function extensionality axiom can be proven from the existence of a directed interval type, consisting of two points and a morphism between them. It can possibly be proven from the directed univalence axiom.

We also assume:

Axiom 3.8.16 (Function extensionality)**.** The function happly is an equivalence: funExt : isEquiv(happly). Hence, for all $f, g :^{=} \prod^{+}_{a:A} C(a)$, we have $(f = g) \simeq (f \sim g)$.

It can still be proven from the existence of an interval type, and presumably also from combining the directed and categorical univalence axioms. A co-inductive proof using only directed function extensionality and the categorical univalence axiom is possible, but co-induction is not obviously valid here.

Isovariant dependent function types are projective limits

Let $A : \mathcal{U}_k$ be a type and $B : A \xrightarrow{+} \mathcal{U}_k$ a covariant type family. Then a function $f : \prod_{a:A} B(a)$ 'is' an element $f(a)$ of every $B(a)$ such that a morphism $\varphi : a \rightsquigarrow_A a'$ implies that $\varphi_*(f(a)) = f(a')$. That sounds a lot like a projective limit.

The universal property of the projective limit $L = \varprojlim_{a:A} B(a)$ with natural projections $p_a : L \xrightarrow{+} B(a)$, is that for any type $X : \mathcal{U}_k$, the functions $f : X \xrightarrow{+} L$ are in bijection with families of functions $f' : \prod^{=}_{a:A} X \xrightarrow{+} B(a)$, where f' is obtained from f by composing with the projections. These families f' should be isovariant in a so as to assure that for every morphism $\varphi : a \rightsquigarrow_A a'$, the following diagram commutes:

$$\begin{array}{ccc} & X & \\ {\scriptstyle f_a}\swarrow & & \searrow{\scriptstyle f_{a'}} \\ B(a) & \xrightarrow[\varphi_*]{} & B(a'). \end{array} \tag{3.162}$$

In directed HoTT, this universal property can be formulated as $\prod^{=}_{X:\mathcal{U}_k} \mathsf{isEquiv}(\kappa(X))$ where

$$\kappa(X) :\equiv f \overset{+}{\mapsto} \left(a \overset{=}{\mapsto} x \overset{+}{\mapsto} p_a(f(x))\right) : (X \xrightarrow{+} L) \xrightarrow{+} \left(\prod_{a:A}^{=} X \xrightarrow{+} B(a)\right). \tag{3.163}$$

For covariant B, the following lemma states that $\prod^{=}_{a:A} B(a)$ with projections $w \overset{+}{\mapsto} w(a)$, is the projective limit $\varprojlim_{a:A} B(a)$:

Lemma 3.8.17. For any type $A :^{=} \mathcal{U}_k$ and any type family $B :^{=} A \xrightarrow{\times} \mathcal{U}_k$, we have

$\prod^{=}_{X:\mathcal{U}_k} \mathsf{isEquiv}(\kappa(X))$ where

$$\kappa :\equiv X \overset{=}{\mapsto} f \overset{+}{\mapsto} \left(a \overset{=}{\mapsto} x \overset{+}{\mapsto} f(x)(a)\right) : \prod_{X:\mathcal{U}_k}^{=} \left(X \overset{+}{\to} \left(\prod_{a:A}^{=} B(a)\right)\right) \overset{+}{\to} \left(\prod_{a:A}^{=} X \overset{+}{\to} B(a)\right). \tag{3.164}$$

Proof. Note that $\kappa(X)$ simply swaps arguments. This is clearly an equivalence. □

3.8.9 Universe types and univalence

In this section, we add the critical element that turns MLTT into HoTT: the univalence axiom ua, which characterizes equality in the universe by stating that a path between types is the same as an equivalence. In directed HoTT, the univalence axiom generalizes in two ways: the directed univalence axiom dua, which characterizes morphisms between types as covariant functions, and the categorical univalence axiom cua, which characterizes paths in *any* type as isomorphisms. The original axiom ua will then be provable from cua and dua. We start by demonstrating (and absolutely not proving), in the symmetric case, that we have the liberty to add such a bold axiom.

There is room for an axiom

We called $\mathbf{0}$ the empty type because it has no constructors. We called $\sum_{a:A} B(a)$ a dependent *pair* type because its only constructor takes two arguments. We claimed that *the* terms of $\mathsf{Fin}(n)$ are $0_n, 1_n, \ldots, (n-1)_n$. Every time, we are silently assuming that the type contains no other terms than those created by its constructors. If we make the same assumption about identity types, then we find that $a =_A b$ only contains $\mathsf{refl}\, a$ when $a \equiv b$ and is empty otherwise; in other words that $a = b$ implies $a \equiv b$. If we look back at most results in this chapter from that perspective, then we see that all of them are trivial. This section demonstrates that the situation is different for identity types than for others. As an example of the situation in other types, we prove the following lemma:

Lemma 3.8.18. Every term of $A + B$ is equal either to $\mathsf{inl}(a)$ for some $a : A$ or to $\mathsf{inr}(b)$ for some $b : B$.

Proof. We have to construct a function

$$f : \prod_{x:A+B} \left(\sum_{a:A} x = \mathsf{inl}(a)\right) + \left(\sum_{b:B} x = \mathsf{inr}(b)\right). \tag{3.165}$$

By induction on x, we can map $\mathsf{inl}\, a$ to $(a, \mathsf{refl}(\mathsf{inl}\, a))$ and $\mathsf{inr}\, b$ to $(b, \mathsf{refl}(\mathsf{inr}\, b))$. □

Note that this does not mean that every $x : A + B$ *is* of the form $\mathsf{inl}\, a$ or $\mathsf{inr}\, b$, but just that there is always a path to an element of that form, which is sufficient to determine the behaviour of x up to equality.

When introducing inductive type families, we noted that the disjoint union of an entire type family is isomorphic to an inductive type. In fact, we can prove the following:

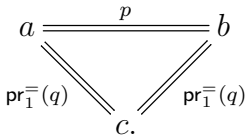
Lemma 3.8.19. Every term of $\sum_{a,b:A}(a =_A b)$ is equal to a term of the form $(c, c, \mathsf{refl}\, c)$ for some $c : A$.

Proof. We have to construct a function of the type

$$f_0 : \prod_{x:\sum_{a,b:A}(a=_A b)} \sum_{c:A} x = (c, c, \mathsf{refl}\, c). \tag{3.166}$$

By repeated induction, we can map $(a, a, \mathsf{refl}\, a)$ to $(a, \mathsf{refl}(a, a, \mathsf{refl}\, a))$. □

So take any $(a, b, p) : \sum_{a,b:A} a =_A b$. Then there exists a c so that $q : (a, b, p) = (c, c, \mathsf{refl}\, c)$. Projecting that path, we find $\mathsf{pr}_1^=(q) : a = c$ and $\mathsf{pr}_2^=(q) : b = c$. The third projection of q shows that $p : a =_A b$ equals $\mathsf{refl}\, c : c =_A c$, but these live in different types, so we have to transport them. Then we conclude that the following diagram commutes:



(3.167)

Again, we do not find that any path *is* reflexivity, but just that any path can be contracted to $\mathsf{refl}\, c$ for some point $c : A$. That is sufficient to define the path's behaviour up to propositional equality. Moreover, this property is perfectly compatible with the groupoid or topological interpretation of types in symmetric HoTT. But it is also compatible with a discrete identity type that coincides with judgemental equality. This indicates that we *need* an axiom somewhere if we want to characterize equality completely. In fact, classical MLTT has a principle called **uniqueness of identity proofs**:

$$\prod_{x,y:A} \prod_{p,q:x=y} p =_{x=y} q. \tag{3.168}$$

HoTT instead has the univalence axiom, which exploits the groupoid structure of the identity type by making it coincide with the natural groupoid structure of the universe.

Univalence in symmetric HoTT

This section is based on [Uni13, §2.10]. Remember that a path $p : A =_{\mathcal{U}_k} B$ implied an equivalence $\mathsf{transport}(p) : A \simeq B$. Voevodsky's univalence axiom states that this is an equivalence:

Axiom 3.8.20 (Univalence)**.** The function $\mathsf{transport}$ is itself an equivalence:

$$\mathsf{ua} : \mathsf{isEquiv}(\mathsf{transport}). \tag{3.169}$$

The left and right inverses $(A \simeq B) \to (A = B)$ of $\mathsf{transport}$ (which are equal), as well as the equivalence $(A \simeq B) \simeq (A = B)$, are also denoted ua.

Lemma 3.8.21. 1. For every equivalence $e : A \simeq B$, we have $\mathsf{transport}(\mathsf{ua}(e)) = e$.

2. For every path $p : A = B$, we have $\mathsf{ua}(\mathsf{transport}(p)) = p$.

3. The function ua behaves well with respect to the equivalence relation structure:

   (a) $\mathsf{ua}(\mathrm{id}_A) = \mathsf{refl}\, A$,

   (b) $\mathsf{ua}(e \circ f) = \mathsf{ua}(e) \circ \mathsf{ua}(f)$,

   (c) $\mathsf{ua}(e^{-1}) = \mathsf{ua}(e)^{-1}$.

Proof. 1. Since ua is the right inverse of $\mathsf{transport}$.

2. Since ua is the left inverse of $\mathsf{transport}$.

3. Prove the corresponding theorems for $\mathsf{transport}$ by induction; then apply $\mathsf{ua}^{=}$ to the proofs and cancel out $\mathsf{ua} \circ \mathsf{transport}$. □

Directed univalence

Recall that the transport function from symmetric HoTT had two generalizations: the directed transport function $\mathsf{dirTransport} : (A \rightsquigarrow_{\mathcal{U}_k} B) \xrightarrow{+} (A \xrightarrow{+} B)$ and the categorical transport function $\mathsf{catTransport} : (a =_A b) \xrightarrow{+} (a \cong_A b)$. The former was based on the idea that morphisms are non-invertible paths and functions are non-invertible equivalences, the latter is based on the idea that morphisms bring the category structure of the universe in symmetric HoTT to arbitrary types in directed HoTT.

We will turn each of these into an equivalence by axiom; then we can prove that the ordinary transport function $\mathsf{transport} : (A =_{\mathcal{U}_k} B) \xrightarrow{+} (A \overset{+}{\simeq} B)$ is also an equivalence. We begin with $\mathsf{dirTransport}$:

Axiom 3.8.22 (Directed univalence)**.** The function $\mathsf{dirTransport}_{A,B} : (A \rightsquigarrow_{\mathcal{U}_k} B) \xrightarrow{+} (A \xrightarrow{+} B)$ is itself an equivalence:

$$\mathsf{dua} : \prod_{A,B:\mathcal{U}_k}^{=} \mathsf{isEquiv}(\mathsf{dirTransport}_{A,B}). \tag{3.170}$$

Lemma 3.8.23. 1. For every function $f : A \xrightarrow{+} B$, we have $\mathsf{dirTransport}(\mathsf{dua}(f)) = f$.

2. For every path $\varphi : A = B$, we have $\mathsf{dua}(\mathsf{dirTransport}(\varphi)) = \varphi$.

3. The function dua behaves well with respect to the category structure:

   (a) $\mathsf{dua}(\mathrm{id}_A) = \mathsf{id}\, A$,

   (b) $\mathsf{dua}(f \circ g) = \mathsf{dua}(f) \circ \mathsf{dua}(g)$.

Proof. 1. Since dua is the right inverse of $\mathsf{dirTransport}$.

2. Since dua is the left inverse of $\mathsf{dirTransport}$.

3. Prove the corresponding theorems for $\mathsf{dirTransport}$ by induction; then apply $\mathsf{dua}^{=}$ to the proofs and cancel out $\mathsf{dua} \circ \mathsf{dirTransport}$. □

3.9 Categorical univalence

3.9.1 Category theory in HoTT

A formulation of category theory within HoTT is given in [Uni13, ch.9]. In this section, we formulate the definition of a category given there, after the necessary preparation. Most of this section is based on [Uni13, §9.1].

Definition 3.9.1. A type P is a **mere proposition** if any two terms of P are equal [Uni13, §3.3]:

$$\mathsf{isProp}(P) :\equiv \prod_{x,y:P} x = y. \tag{3.171}$$

A type S is a **set** if equality of terms of S is a mere proposition [Uni13, §3.1]:

$$\mathsf{isSet}(S) :\equiv \prod_{x,y:S} \prod_{p,q:x=y} p = q. \tag{3.172}$$

We can now define a precategory:

Definition 3.9.2. A **precategory** C is a dependent tuple consisting of:

1. A *type* A_0 of **objects**, also denoted A,
2. For all $a, b : A$, a *set* $a \rightsquigarrow_A b$ of **morphisms**,
3. For each $a : A$, an identity morphism $\mathsf{id}\, a : a \rightsquigarrow a$.
4. For all $a, b, c : A$ a composition function $\circ : (b \rightsquigarrow c) \to (a \rightsquigarrow b) \to (a \rightsquigarrow c)$,
5. A proof that the identity morphisms are left and right unit for composition,
6. A proof that composition of morphisms is associative.

There are no higher coherence laws, as this is a 1-dimensional category whose morphism types are sets.

This already sounds like a good notion of a category. Note that it is somewhat different from a category in classical category theory: a precategory A has a category structure by definition, but on top of that, A_0 carries a higher groupoid structure induced by the identity type family, just as any type does. If this higher groupoid structure has no meaning related to our category structure, we would like to get rid of it. One way to do this is to require that the objects type, too, is a set. Then we arrive at the notion of a

strict category [Uni13, §9.6]. This is somewhat analogous to introducing uniqueness of identity proofs (see section 3.8.9) to get rid of the non-trivial higher groupoid structure in any type. However, the univalence axiom also has its analog: we may require identity to coincide with isomorphism. The definition of isomorphism given in [Uni13, §9.1] is the following:

Definition 3.9.3. A morphism $f : a \rightsquigarrow b$ is an **isomorphism** if there is a morphism $g : b \rightsquigarrow a$ such that $g \circ f = \mathsf{id}\, a$ and $f \circ g = \mathsf{id}\, b$. We write $a \cong b$ for the type of such isomorphisms. (Verbatim from [Uni13, §9.1])

We can spell this out as:

$$(a \cong b) :\equiv \sum_{f:a \rightsquigarrow b} \sum_{g:b \rightsquigarrow a} (g \circ f = 1_a) \times (f \circ g = 1_b)\,. \tag{3.173}$$

Remember that there were problems with the similarly defined notion of quasi-inverse (see section 3.3). It was possible that, for a single function $f : A \to B$, there were multiple distinct proofs that f was invertible in that sense. The reason that the same problem does not occur here, is that we required the morphism types to be sets. In directed HoTT, where the morphism types also have their category structure, we had to take the left-and-right-inverse approach.

Analogous to the situation in the universe, we have a "transport morphism" $a \rightsquigarrow b$ along any path $p : a = b$, and this transport morphism is an isomorphism. We prove right away that $a = b$ implies isomorphism:

Lemma 3.9.4. In a precategory A, identity of objects implies isomorphism: for all $a, b : A$, we have a function

$$\mathsf{catTransport}_{a,b} : (a =_A b) \to (a \cong_A b).$$

Proof. We have to find a function

$$\prod_{a,b:A} \prod_{p:a=b} a \cong b. \tag{3.174}$$

By induction, we can map $a, a, \mathsf{refl}\, a$ to the isomorphism built from $\mathsf{id}\, a$. □

Definition 3.9.5. A **category** is a precategory A such that for all $a, b : A$, the function $\mathsf{catTransport}_{a,b}$ is an equivalence.

That is: in a category, identity and isomorphism coincide. Or: a category is a univalent precategory.

3.9.2 Categorical univalence

At this point, all types in directed HoTT have the structure of a higher precategory: they bear a groupoid structure induced by the identity types, and a category structure induced by the morphism types. The $\mathsf{catTransport}$ function shows that a path always implies an isomorphism, but there is some degree of arbitrariness as to which isomorphisms are paths. Indeed, both judgemental equality $a \equiv b$ and isomorphisms $a \cong_A b$ induce an ∞-groupoid structure and hence a consistent equality type.

In the universe, the directed univalence axiom enforces that the category structure induced by the morphism type corresponds to the one induced by covariant functions. The categorical univalence axiom turns all types from higher precategories into higher categories:

Axiom 3.9.6 (Categorical univalence)**.** The function $\mathsf{catTransport}_{A,a,b} : (a =_A b) \xrightarrow{+} (a \cong_A b)$ is itself an equivalence:

$$\mathsf{cua} : \prod_{A:\mathcal{U}_k}^{=} \prod_{a,b:A}^{=} \mathsf{isEquiv}(\mathsf{catTransport}_{A,B}). \tag{3.175}$$

Lemma 3.9.7. Let $A :^{=} \mathcal{U}_k$ be a type.

1. For every isomorphism $\eta : a \cong_A b$, we have $\mathsf{catTransport}(\mathsf{cua}(\eta)) = \eta$.
2. For every path $p : a =_A b$, we have $\mathsf{cua}(\mathsf{catTransport}(p)) = p$.
3. The function dua behaves well with respect to the category structure:
   (a) $\mathsf{cua}(\mathsf{id}\, a) = \mathsf{refl}\, a$,
   (b) $\mathsf{cua}(\zeta \circ \eta) = \mathsf{cua}(\zeta) \circ \mathsf{cua}(\eta)$,
   (c) $\mathsf{cua}(\eta^{-1}) = \mathsf{cua}(\eta)^{-1}$.

Proof. 1. Since cua is the right inverse of $\mathsf{catTransport}$.

2. Since cua is the left inverse of $\mathsf{catTransport}$.

3. Prove the corresponding theorems for $\mathsf{catTransport}$ by induction; then apply $\mathsf{cua}^{=}$ to the proofs and cancel out $\mathsf{cua} \circ \mathsf{catTransport}$. □

In the proof of Lemma 3.4.13 we have used the injectivity of the function $\mathsf{toMorph}$:

Lemma 3.9.8. For any type $A :^{=} \mathcal{U}_k$, the function $\mathsf{toMorph} : \prod^{=}_{a,b:A}(a =_A b) \xrightarrow{+} (a \rightsquigarrow_A b)$ is injective:

$$\prod_{A:^{=}\mathcal{U}_k}^{=} \prod_{a,b:A}^{=} \prod_{p,q:a=_A b}^{=} (p = q) \overset{+}{\simeq} (\mathsf{toMorph}(p) = \mathsf{toMorph}(q)). \tag{3.176}$$

Proof. As $\mathsf{toMorph}(\mathsf{refl}\, a) \equiv \mathsf{id}\, a$ and $\mathsf{catTransport}(\mathsf{refl}\, a)$ is the identity isomorphism,

we see that $\mathsf{toMorph}(\varphi)$ is just the underlying morphism of $\mathsf{catTransport}(\varphi)$.

($\rightarrow$) This is proven by $\mathsf{toMorph}^{=}$.

($\leftarrow$) Take $s : \mathsf{toMorph}(p) = \mathsf{toMorph}(q)$. We know that $\mathsf{catTransport}(p)$ is an isomorphism with underlying morphism $\mathsf{toMorph}(p)$. This means we have a proof $x : \mathsf{isEquiv}(\mathsf{toMorph}(p))$. Similarly, we have a proof $y : \mathsf{isEquiv}(\mathsf{toMorph}(q))$. Now $s_*(x)$ is in the type $\mathsf{isEquiv}(\mathsf{toMorph}(q))$ and since that is a mere proposition (theorem 3.3.4), we conclude that $s_*(x) = y$ and therefore

$$\mathsf{catTransport}(p) = (\mathsf{toMorph}(p), x) =_{a \cong b} (\mathsf{toMorph}(q), y) = \mathsf{catTransport}(q). \tag{3.177}$$

Applying $\mathsf{cua}^{=}$ yields $p = q$.

We show that applying first the arrow to the left and then the one to the right, yields identity. Since $\mathsf{toMorph} = \mathsf{prl} \circ \mathsf{catTransport}$, we get $\mathsf{toMorph} \circ \mathsf{cua} = \mathsf{prl}$ and thus $\mathsf{toMorph}^{=} \circ \mathsf{cua}^{=} = \mathsf{prl}^{=}$. Now $\mathsf{prl}^{=}$ applied to our proof that $(\mathsf{toMorph}(p), x) =_{a \cong b} (\mathsf{toMorph}(q), y)$, yields the original path s.

We show the converse by induction. The path $\mathsf{refl}\, p$ is mapped to $\mathsf{refl}\, \mathsf{toMorph}(p)$. Then in the construction of the arrow '$\leftarrow$', we can assume that $p \equiv q$, $x \equiv y$ and $s \equiv \mathsf{refl}\, \mathsf{toMorph}(p)$. Now the proof we gave for $s_*(x) = y$ was obtained from the fact that $\mathsf{isEquiv}(q)$ is a mere proposition, so it doesn't necessarily compute to reflexivity. However, one can show that the type $c =_P d$ in a mere proposition P, is also a mere proposition. Then the proof of $s_*(x) = y$ is equal to reflexivity, so that we end up at $\mathsf{refl}\, p$, where we started. □

3.9.3 Univalence in directed HoTT

In this section, we prove the original univalence axiom from categorical and directed univalence. We begin with the following lemma:

Lemma 3.9.9. For any types $A, B :^{=} \mathcal{U}_k$, there is an equivalence $(A \cong_{\mathcal{U}_k} B) \overset{+}{\simeq} (A \overset{+}{\simeq} B)$, which corresponds to $\mathsf{dirTransport}$ and dua for the underlying functions.

Proof. ($\rightarrow$) We build an equivalence of types from the isomorphism $(\varphi, (\mu, p), (\rho, q))$ by applying $\mathsf{dirTransport}$ to φ, μ and ρ and $\mathsf{dirTransport}^{=}$ to $p : \mu \circ \varphi = \mathsf{id}\, A$ and $q : \varphi \circ \rho = \mathsf{id}\, b$.

($\leftarrow$) Similarly, we map $(f, (\ell, p), (r, q))$ to $(\mathsf{dua}(f), (\mathsf{dua}(\ell), \mathsf{dua}^{=}(p)), (\mathsf{dua}(r), \mathsf{dua}^{=}(q)))$.

One can easily show that $\square^{=}$ preserves composition and identity, so that these functions are obviously each others inverse. □

We will call the equivalence from this lemma $\mathsf{dirTransport}$ as well, and its inverse dua. We get:

Lemma 3.9.10. For any types $A, B :^{=} \mathcal{U}_k$, the following diagram commutes:

$$\begin{array}{ccc} A = B & & \\ & \searrow^{\mathsf{catTransport}} & \\ \downarrow{\scriptstyle \mathsf{transport}} & & A \cong_{\mathcal{U}_k} B \\ & \swarrow_{\mathsf{dirTransport}} & \\ A \overset{+}{\simeq} B. & & \end{array} \tag{3.178}$$

Proof. By function extensionality, we have to show

$$\prod_{A,B:\mathcal{U}_k}^{=} \prod_{p:A=B}^{=} \mathsf{transport}(p) = (\mathsf{dirTransport} \circ \mathsf{catTransport})(p). \tag{3.179}$$

By path induction, this becomes

$$\prod_{A:\mathcal{U}_k}^{=} \mathsf{transport}(\mathsf{refl}\, A) = (\mathsf{dirTransport} \circ \mathsf{catTransport})(\mathsf{refl}\, A). \tag{3.180}$$

Now the left hand side computes to $(\mathrm{id}_A, (\mathrm{id}_A, \mathsf{refl}\,\mathrm{id}_A), (\mathrm{id}_A, \mathsf{refl}\,\mathrm{id}_A))$, and the right hand side to

$$\begin{aligned} &\mathsf{dirTransport}(\mathsf{id}\, a, (\mathsf{id}\, a, \mathsf{refl}(\mathsf{id}\, a)), (\mathsf{id}\, a, \mathsf{refl}(\mathsf{id}\, a))) \qquad (3.181)\\ &\equiv (\mathrm{id}_A, (\mathrm{id}_A, \mathsf{refl}\,\mathrm{id}_A), (\mathrm{id}_A, \mathsf{refl}\,\mathrm{id}_A)). \end{aligned}$$

□

These lemmas prove:

Theorem 3.9.11 (Univalence)**.** For any types $A, B :^{=} \mathcal{U}_k$, the function $\mathsf{transport}$ is an equivalence, with inverse $\mathsf{cua} \circ \mathsf{dua}$.

Chapter 4

Exploring semantics and consistency

The Curry-Howard correspondence encodes 'false' as $\mathbf{0}$. Since every proposition is encoded as the type of its proofs, we would like $\mathbf{0}$ to be really empty. In other words, the judgement $\vdash x :^+ \mathbf{0}$ should not be derivable. This is the criterion for directed HoTT to be consistent.

The ambition of this chapter is not to give a proof of consistency, but rather to explore how it could be done. So far, we have been reasoning within directed HoTT – at this point, we want to reason about the theory, so we need to step outside it and pick another formal reasoning system in which we can try to prove the consistency of directed HoTT. Zermelo-Fraenkel set theory with the axiom of choice (ZFC, [Kun13]) is generally the most trusted one, but the gap between type theory and ZFC is fairly big, so that a consistency proof in ZFC would become highly technical and provide little insight in the meaning of judgements in directed HoTT.

The approach I suggest to take is to interpret the language of directed HoTT in a version of Martin-Löf type theory without any axioms about propositional equality. Basically, that means the theory defined by the inference rules for the non-directed case in chapter 2. The translation itself will be defined using elementary reasoning principles that are assumed to exist outside of both directed HoTT and MLTT; these will be called the metatheory. There, we see both theories as a set of formal semantic trees, spanned by the inference rules.

The plan is to define in the metatheory a function $[\![\Box]\!]$ that maps judgements (true or not) from directed HoTT to judgements of MLTT. If for any inference rule in directed HoTT (left), we can derive the rule on the right in MLTT,

$$\frac{\mathcal{P}_1 \qquad \dots \qquad \mathcal{P}_n}{\mathcal{J}} \qquad\qquad \frac{[\![\mathcal{P}_1]\!] \qquad [\![\dots]\!] \qquad [\![\mathcal{P}_n]\!]}{[\![\mathcal{J}]\!]} \tag{4.1}$$

then inconsistency of directed HoTT implies that $[\![\vdash x :^+ \mathbf{0}]\!]$ holds in MLTT. If moreover we can derive in MLTT

$$\frac{[\![\vdash x :^+ \mathbf{0}]\!]}{\vdash x : \mathbf{0}} \tag{4.2}$$

then we can conclude that directed HoTT is consistent if MLTT is.

The first step will be to identify judgements in directed HoTT that clearly mean the same, in order to be able to write every judgement uniquely in the form $\vdash x :^+ A$ for some terms x and A. Secondly, we will define an interpretation function for terms, also denoted $[\![\Box]\!]$, that takes terms from directed HoTT to terms of MLTT. In MLTT we will construct a type Type_k of type labels for the kth universe, with a function $\mathsf{El} : \mathsf{Type}_k \to \mathcal{U}_k$.

Remarkably, these labels will distinguish to some extent between different types from directed HoTT whose elements interpret in the same type in MLTT, but not always. We will set $[\![\mathcal{U}_k]\!] :\equiv \mathsf{Type}_k$.

Untrue judgements in directed HoTT might contain completely nonsensical terms, so we will assume that MLTT has a symbol $\lightning$ that only occurs in false judgements, and we will interpret nonsensical terms to $\lightning$. By passing the desired type $\mathfrak{a} : \mathsf{Type}_k$ to the interpretation function, we can assert that the interpretation of terms is either type-safe or contains the symbol $\lightning$: for any $\mathfrak{a} : \mathsf{Type}_k$, the interpretation $[\![x]\!]_{\mathfrak{a}}$ will either be an element of $\mathsf{El}\,\mathfrak{a}$ or $\lightning$.

The interpretation we give here, relies heavily on co-inductive reasoning *within MLTT*. A more meticulous analysis is needed to establish whether our reasoning is actually valid.

4.1 Basic preparations

Grasping the universes

Suppose, metatheoretically, that directed HoTT is inconsistent. Then we can construct a proof of $\vdash x : \mathbf{0}$. This proof consists of finitely many steps, so it can mention only finitely many universes. Then in order to prove that directed HoTT with an infinite tower of universes is consistent, it suffices to prove that directed HoTT with an arbitrarily large but finite tower of universes is consistent. We will do the latter and construct, for any natural number m in the metatheory, an interpretation of directed HoTT with m universes in MLTT. This is practical, because it allows us to assume that whatever appears to the right of the colon in a judgement of directed HoTT, is an element of $\mathcal{U}_m$ (which is the $(m+1)$th universe).

Judgments, terms and variance

Before we start interpreting judgements, let us reduce the collection of judgements by identifying some of them:

$$
\begin{aligned}
(\Gamma \vdash \mathsf{Ctx}) &\equiv \left(\Gamma \vdash \star :^+ \mathbf{1}\right), \\
(\Gamma \vdash x :^= A) &\equiv \left(\Gamma \vdash \mathsf{gather}\, x :^+ A^{\mathsf{loc}}\right), \\
(\Gamma \vdash x \equiv y :^= A) &\equiv \left(\Gamma \vdash \mathsf{gather}\, x \equiv \mathsf{gather}\, y :^+ A^{\mathsf{loc}}\right), \\
\left(\Gamma \vdash x :^- A\right) &\equiv \left(\Gamma \vdash \mathsf{flip}\, x :^+ A^{\mathsf{op}}\right), \\
\left(\Gamma \vdash x \equiv y :^- A\right) &\equiv \left(\Gamma \vdash \mathsf{flip}\, x \equiv \mathsf{flip}\, y :^+ A^{\mathsf{op}}\right), \\
\left(\Gamma \vdash x :^\times A\right) &\equiv \left(\Gamma \vdash \mathsf{strip}\, x :^+ A^{\mathsf{core}}\right), \\
\left(\Gamma \vdash x \equiv y :^\times A\right) &\equiv \left(\Gamma \vdash \mathsf{strip}\, x \equiv \mathsf{strip}\, y :^+ A^{\mathsf{core}}\right), \\
\left(\Gamma, x :^v A \vdash b[x] :^+ B[x]\right) &\equiv \left(\Gamma \vdash x \overset{v}{\mapsto} b[x] :^+ \prod_{x:A}^{v} B[x]\right), \\
\left(\Gamma, x :^v A \vdash b[x] \equiv c[x] :^+ B[x]\right) &\equiv \left(\Gamma \vdash \left(x \overset{v}{\mapsto} b[x]\right) \equiv \left(x \overset{v}{\mapsto} c[x]\right) :^+ \prod_{x:A}^{v} B[x]\right). \qquad (4.3)
\end{aligned}
$$

Now every judgement is of the form $\vdash x :^+ A$ or $\vdash x \equiv y :^+ A$.

Pre-paths

In 1-dimensional category theory, the localization $\mathcal{A}^{\mathsf{loc}}$ of a category $\mathcal{A}$ has the same objects as $\mathcal{A}$, and its morphisms are the zigzags of $\mathcal{A}$, divided out by an equivalence relation. Indeed, the category with objects from $\mathcal{A}$ and morphisms all zigzags (undivided), clearly has a functor to $\mathcal{A}^{\mathsf{loc}}$ that is constant on objects. Conversely, the category with objects from $\mathcal{A}$ and as $\mathrm{Hom}(a, b)$ the set of zigzags from a to b divided out by the equivalence relation 'true' clearly has a functor from $\mathcal{A}^{\mathsf{loc}}$ that is constant on objects, as it is a groupoid and it has such a functor from $\mathcal{A}$. So A^{loc} is somewhere in between.

Not only is this equivalence relation non-trivial, the concept of an equivalence relation itself is non-trivial in HoTT. Instead of constructing A^{loc} from zigzags, we will construct it using the following type family:

Inductive type family 4.1.1. For any type $A : \mathcal{U}_k$, we define the type family $\square \fallingdotseq_A \square : A \xrightarrow{=} A \xrightarrow{=} \mathcal{U}_k$, which is covariant in A, with the following constructors:

- $\mathsf{prerefl} : \prod^{=}_{a:A} a \fallingdotseq_A a$,
- $a \fallingdotseq_A b$ is isovariant in a,
- $a \fallingdotseq_A b$ is isovariant in b.

Lemma 4.1.2. For any type $A :^{=} \mathcal{U}_k$, we have:

$$\prod_{a,b:A}^{=} (a \fallingdotseq_A b) \overset{+}{\simeq} (\mathsf{gather}\, a =_{A^{\mathsf{loc}}} \mathsf{gather}\, b). \tag{4.4}$$

Proof. ($\to$) The right hand side is isovariant in a and b, so we can apply prepath induction and map $a, a, \mathsf{refl}\, a$ to $\mathsf{refl}(\mathsf{gather}\ a)$.

($\leftarrow$) Using the recursion principle for A^{loc}, we define a type family $C : A^{\mathsf{loc}} \xrightarrow{+} A^{\mathsf{loc}} \xrightarrow{+} \mathcal{U}_k$, so that $C(\mathsf{gather}\, a, \mathsf{gather}\, b) \equiv (a \fallingdotseq_A b)$. Then it suffices to prove

$$\prod_{a',b':A^{\mathsf{loc}}}^{+} (a' =_{A^{\mathsf{loc}}} b') \xrightarrow{+} C(a', b'). \tag{4.5}$$

By path induction, we just need $\prod^{=}_{a':A^{\mathsf{loc}}} : C(a', a')$, and induction on a' leaves $\prod^{=}_{a:A} C(\mathsf{gather}\, a, \mathsf{gather}\, a)$, but here, we can just take $\mathsf{prerefl}\, a$. So we are mapping $a, a, \mathsf{refl}(\mathsf{gather}\ a)$ to $\mathsf{prerefl}\, a$, which is clearly the inverse of the other arrow. □

Lemma 4.1.3. The localization of the universe is a mere proposition.

Proof. Take types $A, B :^{=} \mathcal{U}_k$. The induction principle for $\mathbf{0}$ gives us functions $A \leftarrow \mathbf{0} \to B$, so that $\mathsf{gather}\ A = \mathsf{gather}\ B$. Then the induction principle for A^{loc} allows us to conclude $\prod^{=}_{A',B':\mathcal{U}_k{}^{\mathsf{loc}}} A' = B'$. □

Since one can prove that every mere proposition is also a set (a type in which equality is a mere proposition), we conclude that for any $A, B :^{=} \mathcal{U}_k$, the type $A \fallingdotseq B$ is equivalent to $\mathbf{1}$. We will use this when interpreting pre-paths.

By lack of a good understanding of the notion 'along a pre-path', a good interpretation of pre-path preservation is problematic and at the same time, pre-paths are necessary to interpret A^{loc} and $\sum^{=}_{a:A} C(a)$ (note that $A^{\mathsf{loc}} \overset{+}{\simeq} \sum^{=}_{a:A} \mathbf{1}$). This means that we will be demonstrating consistency of the theory without those types. We will interpret pre-paths as far as we can, clearly exposing where the problem occurs.

Quotes

We will occasionally need to port back terms from MLTT to directed HoTT. When x is a term in MLTT, we will denote by $\langle x \rangle$ a term in directed HoTT that evaluates to x if the proper type is given, and to $\lightning$ otherwise.

Labelling types

It is about time we define Type_k. The sum and product label constructors will already have to refer to El, which is a bit problematic since we would like to define El by recursion on Type_k. The trick is to define them simultaneously in an **inductive-recursive** definition: we give an inductive definition of Type_k that refers to El and as we go, define El recursively. Of course there are restrictions as to what is legal, which are treated in [Dyb00]. A standard application of inductive-recursive definitions is to define a type of labels of *the* universe (as opposed to a formal universe from a different theory, as we are doing here), which is useful, because it provides an induction principle over the universe. An inductive-recursive definition of Type_k together with El is not fundamentally different from that standard application, so it is most likely sound.

Unfortunately, we don't need an inductive type Type_k, but rather a partially co-inductive one. Without going into the details, if a constructor is inductive in an argument, you may not use self-references in that argument when constructing an element, but you may do so when eliminating one. For example, we cannot define a natural number by $n :\equiv \mathsf{succ}\, n$, but we can define $\mathsf{Fin}(n) : \mathcal{U}_0$ by saying $\mathsf{Fin}(\mathsf{succ}\, n) :\equiv \mathsf{Fin}(n) + \mathbf{1}$. Conversely, consider the type $\mathsf{Str}\, A$ of infinite streams of elements of A. It has a constructor $:: : A \to \infty\, \mathsf{Str}\, A \to \mathsf{Str}\, A$ (where the ∞-symbol is pronounced 'delayed' and indicates that the argument is co-inductive). When creating a stream, we are allowed to use self-references:

$$\mathsf{naturals} :\equiv \mathsf{zero} :: \mathsf{addOne}(\mathsf{naturals}), \tag{4.6}$$

but not when eliminating:

$$f(a :: \alpha) :\equiv f(\alpha) \tag{4.7}$$

does not terminate. Co-inductive types are available in the proof assistants Agda and Coq.

So our type Type_k becomes some kind of inductive-co-inductive-recursive type, whose soundness requires more careful analysis. Moreover, from the way we define El, we are assuming that the type formers $\times$ and Σ of the real universe in MLTT are co-inductive as well. Another suspicious feature of this definition is that at one point, we are referring to the interpretation function, which is defined *from the metatheory* by induction over Type.

Note that the Σ-types get variance and the Π-types do not; that localization, opposite and core are included; that an equality label $\doteq$ is present that encodes equality in MLTT, which is possibly more strict than equality in directed HoTT; but bridges, paths, morphisms and pre-paths are not. The constructors of Type_k are selected in such a way that every type from directed HoTT can be *defined* in terms of them.

The delay constructor is useful because it allows us to define $[\![x]\!]$ in terms of $[\![y]\!]$, even when y is not a subterm of x. It is a kind of anti-safety that endangers termination.

Definition 4.1.4. For $k : \mathsf{Fin}(m+2)$ define the type Type_k of **labels for types in** $\mathcal{U}_k$ and a function $\mathsf{El} : \mathsf{Type}_k \to \mathcal{U}_k$ as follows:

- $\mathfrak{U}_\square : \mathsf{Fin}(k) \to \mathsf{Type}_k$, where $\mathsf{El}(\mathfrak{U}_j) = \mathsf{Type}_j$,
- $\mathfrak{o} : \mathsf{Type}_k$, where $\mathsf{El}(\mathfrak{o}) = \mathbf{0}$,
- $\mathfrak{1} : \mathsf{Type}_k$, where $\mathsf{El}(\mathfrak{1}) = \mathbf{1}$,
- $\otimes : \mathsf{Type}_k \to \infty\,\mathsf{Type}_k \to \mathsf{Type}_k$, where $\mathsf{El}(\mathfrak{a} \otimes \mathfrak{b}) = \mathsf{El}(\mathfrak{a}) \times \mathsf{El}(\mathfrak{b})$,
- $\oplus : \mathsf{Type}_k \to \mathsf{Type}_k \to \mathsf{Type}_k$, where $\mathsf{El}(\mathfrak{a} \oplus \mathfrak{b}) = \mathsf{El}(\mathfrak{a}) + \mathsf{El}(\mathfrak{b})$,
- $\mathfrak{N} : \mathsf{Type}_k$, where $\mathsf{El}(\mathfrak{N}) = \mathbb{N}$,
- $\mathfrak{P} : \prod_{\mathfrak{a}:\mathsf{Type}_k} (\mathsf{El}(\mathfrak{a}) \to \mathsf{Type}_k) \to \mathsf{Type}_k$,
  where $\mathsf{El}(\mathfrak{P}^v(\mathfrak{a}, \mathfrak{c})) = \prod_{a:\mathsf{El}(\mathfrak{a})} \mathsf{El}(\mathfrak{c}(a))$,
- $\mathfrak{S} : \mathsf{Var} \to \prod_{\mathfrak{a}:\mathsf{Type}_k} \infty \left(\mathsf{El}\left[\!\!\left[\langle \mathfrak{a} \rangle \xrightarrow{\times} \mathcal{U}_k\right]\!\!\right]\right) \to \mathsf{Type}_k$,
  where $\mathsf{El}(\mathfrak{S}^v(\mathfrak{a}, C)) = \sum_{a:\mathsf{El}(\mathfrak{a})} \mathsf{El}\,[\![\langle C \rangle\,(\langle a \rangle)]\!]$,
- $\doteq : \prod_{S,\mathfrak{c},w} \prod_{\mathfrak{a}:\mathsf{Type}_k} \mathsf{El}(\mathfrak{a}) \to \mathsf{El}(\mathfrak{a}) \to \mathsf{Type}_k$, where $\mathsf{El}(a \doteq_{\mathfrak{a}} b) = (a =_{\mathsf{El}(\mathfrak{a})} b)$,
- $\square^{\mathfrak{op}} : \mathsf{Type}_k \to \mathsf{Type}_k$, where $\mathsf{El}(\mathfrak{a}^{\mathfrak{op}}) = \mathsf{El}(\mathfrak{a})$,
- $\square^{\mathfrak{core}} : \mathsf{Type}_k \to \mathsf{Type}_k$, where $\mathsf{El}(\mathfrak{a}^{\mathfrak{core}}) = \mathsf{El}(\mathfrak{a})$,
- $\square^{\mathfrak{loc}} : \mathsf{Type}_k \to \mathsf{Type}_k$, where $\mathsf{El}(\mathfrak{a}^{\mathfrak{loc}}) = \mathsf{El}(\mathfrak{a})$,
- $\mathsf{delay} : \infty\,\mathsf{Type}_k \to \mathsf{Type}_k$, where $\mathsf{El}(\mathsf{delay}\,\mathfrak{a}) = 1 \times \mathsf{El}(\mathfrak{a})$.

We will write $\mathfrak{a} \Rightarrow \mathfrak{c}$ for the non-dependent function label $\mathfrak{P}(\mathfrak{a}, a \mapsto \mathfrak{c})$.

We will interpret judgements as follows:

$$\begin{aligned} [\![\vdash x :^+ A]\!] \quad &\text{as} \quad \vdash [\![x]\!]_{[\![A]\!]_{\mathfrak{U}_{m+1}}} : \mathsf{El}\,[\![A]\!]_{\mathfrak{U}_{m+1}}\,, \\ [\![\vdash y \equiv y :^+ A]\!] \quad &\text{as} \quad \vdash [\![x]\!]_{[\![A]\!]_{\mathfrak{U}_{m+1}}} \equiv [\![y]\!]_{[\![A]\!]_{\mathfrak{U}_{m+1}}} : \mathsf{El}\,[\![A]\!]_{\mathfrak{U}_{m+1}}\,. \end{aligned} \tag{4.8}$$

4.2 Interpreting terms

In this section, we sketch how we will interpret terms from directed HoTT. The interpretation function $[\![t]\!]_{\mathfrak{a}}$ takes as arguments a term t and a type label $\mathfrak{a} : \mathsf{Type}_k$, and should output an MLTT-term of type $\mathsf{El}\,\mathfrak{a}$, or $\lightning$.

A problem when interpreting HoTT (in particular when you want to implement it as a programming language), is that it contains an axiom: the terms ua and funExt have no computation rules, and it is not immediately clear what they are, and the fact that we can nevertheless prove quite a few theorems about them, makes interpretation a delicate matter. In directed HoTT, this problem multiplies: we have cua, dua, funExt, $\mathsf{dirFunExt}$ and a truckload of axioms for bridges.

Back in the days before the univalence axiom, funExt was already posing a similar problem in MLTT: unless you want to identify any two pointwise equal functions judgementally (which complicates automatic type-checking), $\mathsf{funExt}(H) : f = g$ might be an equality between different functions, in which case transport along $\mathsf{funExt}(H)$ is not trivial to understand or, in programming applications, to compute. In [AMS07], Altenkirch, McBride and Swierstra present a type theory which has function extensionality, but also computing transport (which is as good as the induction principle). The key idea is to define $a =_A b$ by induction on the type A, so that you can implement the transport function, also by induction on A. For example, they define equality of pairs as componentwise equality and equality of functions as pointwise equality. This provides an easy way to implement funExt: it is the identity function. We will do the same, not only for the identity type, but also for bridges, morphisms and pre-paths. We start with the identity type.

4.2.1 Interpreting types

Identity types

The categorical univalence axiom suggests that we should define $a =_A b$ as $a \cong_A b$. Since isomorphism is defined in terms of equality, this seems to be a problem, but fortunately, we are doing co-induction, so it isn't:

$$[\![a =_A b]\!]_{\mathfrak{U}_i} :\equiv \mathsf{delay}\,[\![a \cong_A b]\!]_{\mathfrak{U}_i}\,. \tag{4.9}$$

Morphism types

Much like in [AMS07], we will define $[\![a \rightsquigarrow_A b]\!]_{\mathfrak{U}_i}$ by induction on $[\![A]\!]_{\mathfrak{U}_i}$. Since any term in MLTT can be computed to a unique normal form [ML84] (this property is called strong normalization), we may assume that $[\![A]\!]_{\mathfrak{U}_i}$ is either $\lightning$ or an instance of one of the constructors of Type_i. We define the interpretation of morphisms as follows:

$[\![A]\!]_{\mathfrak{U}_i}$	$[\![a \rightsquigarrow_A b]\!]_{\mathfrak{U}_i}$
$\lightning$	$\lightning$
$\mathfrak{U}_j$	$\mathsf{delay}\left[\!\!\left[a \xrightarrow{+} b\right]\!\!\right]$
$\mathfrak{o}, \mathfrak{1}, \mathfrak{N}$	$[\![a]\!] \doteq [\![b]\!]$
$\mathfrak{c} \otimes \mathfrak{d}$	$\mathsf{delay}\left([\![\mathsf{prl}\,a \rightsquigarrow_{\langle\mathfrak{c}\rangle} \mathsf{prl}\,b]\!] \otimes [\![\mathsf{prr}\,a \rightsquigarrow_{\langle\mathfrak{d}\rangle} \mathsf{prr}\,b]\!]\right)$
$\mathfrak{c} \oplus \mathfrak{d}$	by induction on $[\![a]\!]_{\mathfrak{c}\oplus\mathfrak{d}}$ and $[\![b]\!]_{\mathfrak{c}\oplus\mathfrak{d}}$
	$\mathsf{inl}\,a', \mathsf{inl}\,b' \mapsto \mathsf{delay}\,[\![\langle a'\rangle \rightsquigarrow_{\mathfrak{c}} \langle b'\rangle]\!]$

$[\![A]\!]_{\mathfrak{U}_i}$	$[\![a \rightsquigarrow_A b]\!]_{\mathfrak{U}_i}$
	$\mathsf{inr}\, a', \mathsf{inr}\, b' \mapsto \mathsf{delay}\, [\![\langle a' \rangle \rightsquigarrow_{\mathfrak{d}} \langle b' \rangle]\!]$
	otherwise $\mapsto \mathfrak{o}$
$\mathfrak{P}(\mathfrak{c}, \mathfrak{d})$	$\mathsf{delay}\, \mathfrak{P}\left(\mathfrak{c}, (x : \mathsf{El}\, \mathfrak{c}) \mapsto [\![a(\langle x \rangle) \rightsquigarrow_{\langle \mathfrak{d}(x) \rangle} b(\langle x \rangle)]\!]\right)$
	As $\mathsf{El}\, \mathfrak{P}(\mathfrak{c}, \mathfrak{d})$ contains ordinary functions with no variance behaviour, there is nothing more to say.
$\mathfrak{S}^+(\mathfrak{c}, D)$	$\mathsf{delay}\, \mathfrak{S}^+\left([\![\mathsf{prl}\, a \rightsquigarrow_{\langle \mathfrak{c} \rangle} \mathsf{prl}\, b]\!], \left[\!\left[\varphi \overset{\times}{\mapsto} \beta_*(\mathsf{prr}\, a) \rightsquigarrow_{\mathsf{Rel}_D(\beta)} \beta^*(\mathsf{prr}\, b)\right]\!\right]\right)$
	where $\beta :\equiv \mathsf{toBrid}(\varphi)$
$\mathfrak{S}^-(\mathfrak{c}, D)$	$\mathsf{delay}\, \mathfrak{S}^+\left([\![\mathsf{prl}\, b \rightsquigarrow_{\langle \mathfrak{c} \rangle} \mathsf{prl}\, a]\!], \left[\!\left[\varphi \overset{\times}{\mapsto} \beta^*(\mathsf{prr}\, a) \rightsquigarrow_{\mathsf{Rel}_D(\beta)} \beta_*(\mathsf{prr}\, b)\right]\!\right]\right)$
	where $\beta :\equiv \mathsf{toBrid}(\varphi)$
$\mathfrak{S}^\times(\mathfrak{c}, D)$	$\mathsf{delay}\, \mathfrak{S}^+\left([\![\mathsf{prl}\, a =_{\langle \mathfrak{c} \rangle} \mathsf{prl}\, b]\!], \left[\!\left[p \overset{\times}{\mapsto} \beta_*(\mathsf{prr}\, a) \rightsquigarrow_{\mathsf{Rel}_D(\beta)} \beta^*(\mathsf{prr}\, b)\right]\!\right]\right)$
	where $\beta :\equiv \mathsf{toBrid}(\mathsf{toMorph}(p))$
$\mathfrak{S}^=(\mathfrak{c}, D)$	$\mathsf{delay}\, \mathfrak{S}^+\left([\![\mathsf{prl}\, a \eqsim_{\langle \mathfrak{c} \rangle} \mathsf{prl}\, b]\!], \left[\!\left[p \overset{\times}{\mapsto} \mathbf{problematic}\right]\!\right]\right)$
	Note that the variance of the sums on the right is not a typo.
$x \doteq_X y$	$[\![a]\!] \doteq [\![b]\!]$
$\mathfrak{a}^{\mathrm{op}}$	$\mathsf{delay}\, [\![\mathsf{unflip}\, b \rightsquigarrow_{\langle \mathfrak{a} \rangle} \mathsf{unflip}\, a]\!]$
$\mathfrak{a}^{\mathrm{core}}$	$\mathsf{delay}\, [\![\mathsf{unstrip}\, a =_{\langle \mathfrak{a} \rangle} \mathsf{unstrip}\, b]\!]$
$\mathfrak{a}^{\mathrm{loc}}$	Since $\mathsf{El}\, \mathfrak{a} = \mathsf{El}\, \mathfrak{a}^{\mathrm{loc}}$, we can interpret terms of $\mathfrak{a}^{\mathrm{loc}}$ and plug them into $\mathfrak{a}$:
	$\mathsf{delay}\, [\![\langle [\![a]\!]_{\mathfrak{a}^{\mathrm{loc}}} \rangle \eqsim_{\langle \mathfrak{a} \rangle} \langle [\![b]\!]_{a^{\mathrm{loc}}} \rangle]\!]$
$\mathsf{delay}\, \mathfrak{a}$	$\mathsf{delay}\, [\![\mathsf{prr}\, a \rightsquigarrow_{\langle \mathfrak{a} \rangle} \mathsf{prr}\, b]\!]$

Bridge types

We define the interpretation of bridges through the following table:

$[\![A]\!]_{\mathfrak{U}_i}$	$[\![a \frown_A b]\!]_{\mathfrak{U}_i}$
$\lightning$	$\lightning$
$\mathfrak{U}_j$	$\mathsf{delay}\, \left[\!\left[\sum^+_{X : \mathcal{U}_j} (a \xrightarrow{+} X) \times (b \xrightarrow{+} X)\right]\!\right]$
$\mathfrak{o}, \mathfrak{1}, \mathfrak{N}$	$[\![a]\!] \doteq [\![b]\!]$
$\mathfrak{c} \otimes \mathfrak{d}$	$\mathsf{delay}\, \left([\![\mathsf{prl}\, a \frown_{\langle \mathfrak{c} \rangle} \mathsf{prl}\, b]\!] \otimes [\![\mathsf{prr}\, a \frown_{\langle \mathfrak{d} \rangle} \mathsf{prr}\, b]\!]\right)$
$\mathfrak{c} \oplus \mathfrak{d}$	by induction on $[\![a]\!]_{\mathfrak{c} \oplus \mathfrak{d}}$ and $[\![b]\!]_{\mathfrak{c} \oplus \mathfrak{d}}$
	$\mathsf{inl}\, a', \mathsf{inl}\, b' \mapsto \mathsf{delay}\, [\![\langle a' \rangle \frown_{\mathfrak{c}} \langle b' \rangle]\!]$
	$\mathsf{inr}\, a', \mathsf{inr}\, b' \mapsto \mathsf{delay}\, [\![\langle a' \rangle \frown_{\mathfrak{d}} \langle b' \rangle]\!]$
	otherwise $\mapsto \mathfrak{o}$
$\mathfrak{P}(\mathfrak{c}, \mathfrak{d})$	$\mathsf{delay}\, \mathfrak{P}\left(\mathfrak{c}, (x : \mathsf{El}\, \mathfrak{c}) \mapsto [\![a(\langle x \rangle) \frown_{\langle \mathfrak{d}(x) \rangle} b(\langle x \rangle)]\!]\right)$
$\mathfrak{S}^{+/-/\times}(\mathfrak{c}, D)$	$\mathsf{delay}\, \mathfrak{S}^+\left([\![\mathsf{prl}\, a \frown_{\langle \mathfrak{c} \rangle} \mathsf{prl}\, b]\!], \left[\!\left[\beta \overset{\times}{\mapsto} \beta_*(\mathsf{prr}\, a) \rightsquigarrow_{\mathsf{Rel}_D(\beta)} \beta^*(\mathsf{prr}\, b)\right]\!\right]\right)$
$\mathfrak{S}^=(\mathfrak{c}, D)$	$\mathsf{delay}\, \mathfrak{S}^+\left([\![\mathsf{prl}\, a \eqsim_{\langle \mathfrak{c} \rangle} \mathsf{prl}\, b]\!], \left[\!\left[p \overset{\times}{\mapsto} \mathbf{problematic}\right]\!\right]\right)$
$x \doteq_X y$	$[\![a]\!] \doteq [\![b]\!]$
$\mathfrak{a}^{\mathrm{op}}$	$\mathsf{delay}\, [\![\mathsf{unflip}\, a \frown_{\langle \mathfrak{a} \rangle} \mathsf{unflip}\, b]\!]$
$\mathfrak{a}^{\mathrm{core}}$	$\mathsf{delay}\, [\![\mathsf{unstrip}\, a \frown_{\langle \mathfrak{a} \rangle} \mathsf{unstrip}\, b]\!]$
$\mathfrak{a}^{\mathrm{loc}}$	Since $\mathsf{El}\, \mathfrak{a} = \mathsf{El}\, \mathfrak{a}^{\mathrm{loc}}$, we can interpret terms of $\mathfrak{a}^{\mathrm{loc}}$ and plug them into $\mathfrak{a}$:
	$\mathsf{delay}\, [\![\langle [\![a]\!]_{\mathfrak{a}^{\mathrm{loc}}} \rangle \eqsim_{\langle \mathfrak{a} \rangle} \langle [\![b]\!]_{a^{\mathrm{loc}}} \rangle]\!]$
$\mathsf{delay}\, \mathfrak{a}$	$\mathsf{delay}\, [\![\mathsf{prr}\, a \frown_{\langle \mathfrak{a} \rangle} \mathsf{prr}\, b]\!]$

Pre-path types

We define the interpretation of pre-paths through the following table:

$[\![A]\!]_{\mathfrak{U}_i}$	$[\![a \frown_A b]\!]_{\mathfrak{U}_i}$
$\lightning$	$\lightning$
$\mathfrak{U}_j$	$\mathfrak{1}$
$\mathfrak{o}, \mathfrak{1}, \mathfrak{N}$	$[\![a]\!] \doteq [\![b]\!]$
$\mathfrak{c} \otimes \mathfrak{d}$	$\mathsf{delay}\,\big([\![\mathsf{prl}\,a \mathrel{\fallingdotseq}_{\langle\mathfrak{c}\rangle} \mathsf{prl}\,b]\!] \otimes [\![\mathsf{prr}\,a \mathrel{\fallingdotseq}_{\langle\mathfrak{d}\rangle} \mathsf{prr}\,b]\!]\big)$
$\mathfrak{c} \oplus \mathfrak{d}$	by induction on $[\![a]\!]_{\mathfrak{c}\oplus\mathfrak{d}}$ and $[\![b]\!]_{\mathfrak{c}\oplus\mathfrak{d}}$
	$\mathsf{inl}\,a', \mathsf{inl}\,b' \mapsto \mathsf{delay}\,[\![\langle a'\rangle \mathrel{\fallingdotseq}_{\mathfrak{c}} \langle b'\rangle]\!]$
	$\mathsf{inr}\,a', \mathsf{inr}\,b' \mapsto \mathsf{delay}\,[\![\langle a'\rangle \mathrel{\fallingdotseq}_{\mathfrak{d}} \langle b'\rangle]\!]$
	otherwise $\mapsto \mathfrak{o}$
$\mathfrak{P}(\mathfrak{c}, \mathfrak{d})$	$\mathsf{delay}\,\mathfrak{P}\,\big(\mathfrak{c}, (x : \mathsf{El}\,\mathfrak{c}) \mapsto [\![a(\langle x\rangle) \mathrel{\fallingdotseq}_{\langle\mathfrak{d}(x)\rangle} b(\langle x\rangle)]\!]\big)$
$\mathfrak{S}^v(\mathfrak{c}, D)$	$\mathsf{delay}\,\mathfrak{S}^+\left([\![\mathsf{prl}\,a \mathrel{\fallingdotseq}_{\langle\mathfrak{c}\rangle} \mathsf{prl}\,b]\!]\,, \left[\!\!\left[p \overset{\times}{\mapsto} \textbf{problematic}\right]\!\!\right]\right)$
$x \doteq_X y$	$[\![a]\!] \doteq [\![b]\!]$
$\mathfrak{a}^{\mathfrak{op}}$	$\mathsf{delay}\,[\![\mathsf{unflip}\,a \mathrel{\fallingdotseq}_{\langle\mathfrak{a}\rangle} \mathsf{unflip}\,b]\!]$
$\mathfrak{a}^{\mathfrak{core}}$	$\mathsf{delay}\,[\![\mathsf{unstrip}\,a \mathrel{\fallingdotseq}_{\langle\mathfrak{a}\rangle} \mathsf{unstrip}\,b]\!]$
$\mathfrak{a}^{\mathfrak{loc}}$	Since $\mathsf{El}\,\mathfrak{a} = \mathsf{El}\,\mathfrak{a}^{\mathfrak{loc}}$, we can interpret terms of $\mathfrak{a}^{\mathfrak{loc}}$ and plug them into $\mathfrak{a}$:
	$\mathsf{delay}\,[\![\langle [\![a]\!]_{\mathfrak{a}^{\mathfrak{loc}}}\rangle \mathrel{\fallingdotseq}_{\langle\mathfrak{a}\rangle} \langle [\![b]\!]_{a^{\mathfrak{loc}}}\rangle]\!]$
$\mathsf{delay}\,\mathfrak{a}$	$\mathsf{delay}\,[\![\mathsf{prr}\,a \mathrel{\fallingdotseq}_{\langle\mathfrak{a}\rangle} \mathsf{prr}\,b]\!]$

Function types

A covariant function $f : A \xrightarrow{+} C$ is a function $f : A \to C$ that preserves morphisms, bridges and pre-paths in a way that commutes with the weakening of morphisms to bridges and bridges to pre-paths. Thus, we interpret $A \xrightarrow{+} C$ as:

- the type label for ordinary functions $A \to C$: $[\![A]\!] \Rightarrow [\![C]\!]$,
- times the claim that f preserves morphisms: $\left[\!\!\left[\left(\prod^{=}_{a,b:A} \prod^{+}_{\varphi:a\rightsquigarrow b} f(a) \rightsquigarrow f(b)\right)^{\mathsf{core}}\right]\!\!\right]$,
- times the claim that f preserves bridges: $\left[\!\!\left[\left(\prod^{=}_{a,b:A} \prod^{?}_{\varphi:a\frown b} f(a) \frown f(b)\right)^{\mathsf{core}}\right]\!\!\right]$,
- times the claim that f preserves pre-paths: $\left[\!\!\left[\left(\prod^{=}_{a,b:A} \prod^{+}_{\varphi:a\mathrel{\fallingdotseq} b} f(a) \mathrel{\fallingdotseq} f(b)\right)^{\mathsf{core}}\right]\!\!\right]$,
- times the claim that this all happens in a well-behaved way (again taking the core of the proposition).

The reason we have to take the core of all the additional information, is that it is available isovariantly in the theory. Now if we have a morphism $f \rightsquigarrow g$, then we have a morphism between the proofs of covariance of f and g *in the core*, so that the proofs are equal (after transport).

So $\left[\!\!\left[A \xrightarrow{+} B\right]\!\!\right]_{\mathfrak{U}_i}$ is interpreted as:

$$([\![A]\!] \Rightarrow [\![C]\!]) \otimes \left[\!\!\left[\left(\prod^{=}_{a,b:A} \prod^{+}_{\varphi:a\rightsquigarrow b} f(a) \rightsquigarrow f(b)\right)^{\mathsf{core}}\right]\!\!\right]$$

$$\otimes \left[\!\!\left[\left(\prod_{a,b:A}^{=} \prod_{\beta:a\frown b}^{?} f(a) \frown f(b) \right)^{\mathsf{core}} \right]\!\!\right] \otimes \left[\!\!\left[\left(\prod_{a,b:A}^{=} \prod_{p:a\fallingdotseq b}^{+} f(a) \fallingdotseq f(b) \right)^{\mathsf{core}} \right]\!\!\right]$$
$$\otimes [\![\mathsf{wellBehaved}^{\mathsf{core}}]\!], \tag{4.10}$$

or as $\lightning$ if A or B are not types.

We can write similar interpretations for function types of different variances. For dependent functions, we state all the good-behaviour properties in the type $\mathsf{Rel}_C(\beta)$, where β is a bridge obtained from the given bridge or morphism. Again, pre-path preservation for dependent functions is problematic because there is no analogue of $\mathsf{Rel}_C(p)$ for pre-paths p.

Other types

The interpretation of most other types is trivial:

$$[\![A \times B]\!] :\equiv [\![A]\!] \otimes [\![B]\!], \qquad [\![A + B]\!] :\equiv [\![A]\!] \oplus [\![B]\!],$$
$$[\![\mathbf{0}]\!] :\equiv \mathfrak{o}, \qquad [\![\mathbf{1}]\!] :\equiv \mathfrak{1}, \qquad [\![\mathbb{N}]\!] :\equiv \mathfrak{N},$$
$$[\![A^{\mathsf{op}}]\!] :\equiv [\![A]\!]^{\mathfrak{op}}, \qquad [\![A^{\mathsf{core}}]\!] :\equiv [\![A]\!]^{\mathfrak{core}}, \qquad [\![A^{\mathsf{loc}}]\!] :\equiv [\![A]\!]^{\mathfrak{loc}},$$
$$\left[\!\!\left[\sum_{x:A}^{v} C(x) \right]\!\!\right]_{\mathfrak{U}_k} :\equiv \mathfrak{S}^{v} \left([\![A]\!], [\![C]\!]_{[\![A^{\times} \to \mathcal{U}_k]\!]} \right).$$

4.2.2 Functorial behaviour

There is very much to say about the interpretation of the functorial behaviour of all functions. We just consider two cases.

The identity type maps morphisms to bridges

As $x \overset{\times}{\mapsto} a =_A x$ is an invariant function, a morphism $\varphi : b \rightsquigarrow c$ should yield a bridge $(a = b) \frown (a = c)$. In particular, if $\beta \equiv \mathsf{toBrid}(\varphi)$, we get

$$(a = b) \xrightarrow{\beta_*} \mathsf{Rel}_{a=\square}(\beta) \xleftarrow{\beta^*} (a = c). \tag{4.11}$$

From chapter 3, we know that $p : a = b$ and $q : a = c$ are transported to equal elements of $\mathsf{Rel}_{a=\square}(\beta)$ precisely if they are equal along φ, which we showed is equivalent to saying that the following diagram commutes:

$$\begin{array}{ccc} a & \overset{p}{=\!=} & b \\ {\scriptstyle \mathsf{id}\, a}\big\downarrow & & \big\downarrow{\scriptstyle \varphi} \\ a & \underset{q}{=\!=} & c. \end{array} \tag{4.12}$$

It is not immediately clear what the type $\mathsf{Rel}_{a=\square}(\beta)$ could look like. However, the interpretation of $a =_A b$ is the same as that of $a \cong_A b$, and we can spell that out:

$$\sum_{\varphi:a\rightsquigarrow b}^{+} \left(\sum_{\mu:b\rightsquigarrow a}^{+} \mu \circ \varphi =_{a\rightsquigarrow a} \mathsf{id}\, a \right) \times \left(\sum_{\rho:b\rightsquigarrow a}^{+} \varphi \circ \rho =_{b\rightsquigarrow b} \mathsf{id}\, b \right). \tag{4.13}$$

This expression only uses a and b co- and contravariantly, never invariantly, so we could actually construct a bridge from $a \cong_A b$ to $a \cong_A c$ *within* directed HoTT! So we can interpret the functorial behaviour of $a = \square$ as that of $a \cong \square$.

The morphism type preserves bridges

As $a \rightsquigarrow_A c$ is contravariant in a and covariant in c, it should preserve bridges on both sides. When we consider how bridges and morphisms are interpreted in the different base types from Type, we see that there are only a few cases where this is a remarkable fact: the core, the universe and Σ-types.

In the core, a morphism $a \rightsquigarrow_{A^{\mathsf{core}}} c$ has the same interpretation as a path $\mathsf{unflip}\, a =_A \mathsf{unflip}\, c$, and a bridge $a \frown_{A^{\mathsf{core}}} b$ is just a bridge $\mathsf{unflip}\, a \frown_A \mathsf{unflip}\, b$, so basically it is sufficient to show that paths preserve bridges. But paths are isomorphisms, which are defined as morphisms, so we can delegate the issue from A^{core} to A.

For Σ-types, the problem for the first component can simply be delegated to that component's type (or its opposite/core/localization). As for the second component, the transporting complicates the situation. I have no solution for this case.

For function types, it is clear that a cospan $B \xrightarrow{f} X \xleftarrow{g} C$ leads to a cospan from $A \xrightarrow{+} B$ to $A \xrightarrow{+} C$, so (non-dependent) function types preserve bridges in the codomain. In the domain, the cospan is mapped to a span

$$(B \xrightarrow{+} A) \xleftarrow{\square \circ f} (X \xrightarrow{+} A) \xrightarrow{\square \circ g} (C \xrightarrow{+} A). \tag{4.14}$$

Then functions $h : B \xrightarrow{+} A$ and $k : C \xrightarrow{+} A$ are considered equal if there is a function $\ell : X \xrightarrow{+} A$ so that $h = \ell \circ f$ and $k = \ell \circ g$. This is the same as saying that h and k take identical values on arguments that are assumed to be identical along the cospan of domains. If we want to be able to transport, we can take the pushout of the span of functions, or equivalently the pullback of the cospan of domains. Pushouts and pullbacks could be constructed using the (problematic) $\sum^{=}$-types and an additional type that represents the index category of spans.

4.3 Interpreting axioms

Function extensionality As we have interpreted a morphism between functions as a morphism on every point and an equality of the functoriality proofs (which, after transport, should amount to saying that all the necessary diagrams commute), dirFunExt should now be implementable. As a path is an isomorphism, which consists of infinitely many morphisms, funExt is can probably be interpreted co-inductively in terms of the interpretation of dirFunExt.

Categorical univalence By interpreting paths as isomorphisms, the interpretation of catTransport has probably effectively become the identity function. Then cua can be interpreted as a proof that the identity function is an equivalence.

Directed univalence Similarly, we have interpreted morphisms between types as covariant functions, so dirTransport should also have become the identity function, and then dua, too, can be interpreted as a proof that the identity is an equivalence.

The bridge axioms A type per type analysis of whether and how bridges can be preserved, could (in)validate the list of bridge axioms from section 3.5.

Morphism induction Paths, morphisms, and pre-paths are the only types T where the elimination rule does not automatically compute for all elements of $\mathsf{El}\,[\![T]\!]$. (For the bridge type, it is disputable which rules are axioms and which ones are elimination rules.) Therefore, they are a bit like axioms and it is interesting to consider their interpretation. We only treat the covariant morphism induction principle here. We can interpret it quite easily as follows:

Given a type family $C : A \xrightarrow{-} A \xrightarrow{+} A$ and a function $f : \prod^{=}_{x:A} C(x)(x)$, we interpret $\mathsf{ind}^{+}_{\rightsquigarrow}(C, f)(a, b, \varphi)$ as $(C(a)^{\rightsquigarrow}(\varphi))_*\,(f(a)) : C(a)(b)$.

Higher order variance In this chapter, we have completely ignored the existence of functions with complicated higher order variance, such as $A \mapsto A^{\mathsf{op}}$. They are likely to make the problem more complicated and subtle.

4.4 Conclusion

In this chapter, we made a first few steps at interpreting directed HoTT in MLTT. We encountered several issues on the way:

- A high degree of co-inductiveness, as well as a co-inductive-inductive-recursive type are involved.
- An interpretation of morphisms in the localization and the $\sum^{=}$-type is problematic, due to the difficulty of understanding the notion 'along a pre-path'. In particular, a pre-path in the universe contains no information.
- The functorial behaviour of the morphism type in $\sum_{x:A} C(x)$ is problematic when C uses x invariantly, due to the non-composability of bridges. If we allow composition of bridges, then the notion of equality of bridges becomes difficult.
- Higher order variance probably interacts poorly with this interpretation. It is probably better to have native higher order variance annotations in directed HoTT.

Nevertheless, this interpretation makes the univalence axioms very plausible and is promising when it comes to the function extensionality axioms. Moreover, it suggests that we could add an inference rule to the theory that takes:

- An invariant function $f : \prod^{\times}_{x:A} C(x)$,
- A proof that it preserves morphisms, $\prod^{=}_{a,b:A} \prod^{+}_{\varphi:a\rightsquigarrow b} \varphi_*(f(a)) \rightsquigarrow_{\mathsf{Rel}_C(\varphi)} \varphi^*(f(b))$,
- A proof that this preservation is well-behaved with respect to the already present preservation of paths and bridges,

and yields a covariant function g so that $g_\times = f$. Of course there would be similar rules for other variances.

Chapter 5

Conclusion

5.1 Morphisms and morphisms

In symmetric HoTT, the univalence axiom implies a more general these that isomorphic structures are equal. In directed HoTT, the directed univalence axiom leads to the these that a domain specific morphism implies a type theoretic morphism. This section demonstrates the these by proving informally that a morphism between groups is the same as a group morphism.

Definition 5.1.1. An (unordered) **set** is a bridgeless groupoid with uniqueness of identity proofs:

$$\mathsf{isSet}(A) :\equiv \mathsf{isBridgelessGrpd}(A) \times \prod_{a,b:A}^{=} \prod_{p,q:a=b}^{=} p =_{a=b} q. \tag{5.1}$$

Definition 5.1.2. A **group** $\mathcal{G} : \mathsf{Grp}$ is a (covariant) dependent tuple consisting of:

- A type G,
- A proof that G is a set,
- A binary operation $* : G \xrightarrow{4} G \xrightarrow{4} G$,
- A neutral element $e : G$,
- An inversion function $\square^{-1} : G \xrightarrow{4} G$,
- A proof of associativity: for all $x, y, z :^4 G$, we have $x * (y * z) = (x * y) * z$,
- A proof of neutrality of e: for all $x :^4 G$, we have $e * x = x$ and $x * e = x$,
- A proof that inversion works: for all $x :^4 G$, we have $x * x^{-1} = e$ and $x^{-1} * x = e$.

Definition 5.1.3. Let $\mathcal{G}, \mathcal{H} : \mathsf{Grp}$ be groups. A **morphism of groups** is a dependent tuple consisting of:

- A function $f : G \xrightarrow{4} H$,
- For all $x, y :^4 G$, a proof that $f(x *_G y) =_H f(x) *_H f(y)$,
- A proof that $f(e_G) =_H e_H$,
- For all $x :^4 G$, a proof that $f(x^{-1}) =_H f(x)^{-1}$.

Proposition 5.1.4. A group morphism is a morphism between groups:

$$\prod_{\mathcal{G},\mathcal{H}:\mathsf{Grp}}^{=} (\mathcal{G} \rightsquigarrow_{\mathsf{Grp}} \mathcal{H}) \overset{+}{\simeq} \mathsf{GrpMph}(\mathcal{G}, \mathcal{H}). \tag{5.2}$$

Plausibility argument. We will assume that $\mathsf{isSet}(A)$ is always a mere proposition and, in particular, a mere proposition. From the characterization of morphisms in the $\sum^+$-type, morphisms along morphisms and the directed transport lemma, as well as a few easily provable facts about transport, we know that a morphism $\mathcal{G} \rightsquigarrow_{\mathsf{Grp}} \mathcal{H}$ boils down to:

- A function $f : G \xrightarrow{4} H$,
- A proof that the proofs of G and H being sets, are related along $\mathsf{dua}(f)$, which is trivially true as $\mathsf{isSet}(A)$ is always a mere proposition,
- A proof that $f(\square *_G \square) =_H f(\square) *_H f(\square)$,
- A proof that $f(e_G) =_H e_H$,
- A proof that, for all $x :^4 G$, $f(x^{-1}) =_H f(x)^{-1}$,
- A proof that the associativity, neutrality and inversion proofs are compatible along $\mathsf{dua}(f)$, which is trivially true since all of them are mere propositions.

Now all we need to do is throw out those components that are trivially true (equivalent to $\mathbf{1}$) and apply $\mathsf{happly}/\mathsf{funExt}$ to some others. □

It should be noted that some of the details may actually pose substantial problems. Moreover, morphisms seem less well-behaved when we don't require types to be groupoidal. Consider for example the type $\sum^+_{A:\mathcal{U}_k} A$ of types with a designated element. A morphism $(A, a) \rightsquigarrow (B, b)$ consists of a function $f : A \xrightarrow{+} B$ and a *morphism* $f(a) \rightsquigarrow_B b$. That is peculiar, we would have expected a path $f(x) =_B b$. This can be solved by taking the core: perhaps we should instead have considered the type $\sum^+_{A:\mathcal{U}_k} A^{\mathsf{core}}$. The question is whether this quick-and-dirty solution is well-behaved.

5.2 Applications

When using directed HoTT as a foundation for mathematics, you don't need to define morphisms any more: the definition of a vector space gives you linear maps, the definition of a group gives you group morphisms, etc. Moreover, in order to prove that a property of e.g. a group element is preserved under morphisms, you just need to show that the property is covariant in the group.

Infinite dimensional category theory is a subject that is not yet completely understood. If directed HoTT can be shown to be consistent relative to something other than infinite category theory, then it may be a useful tool in studying that subject, providing a constructive and potentially computer verified way to work with infinite dimensional categories.

In computer science, it could provide a safe and nearly automatic way to keep track of naturality. This may allow for quick implementations of non-trivial functions and properties. Consider monads, for example. A nice property of dependent type theory is that it allows you to implement the monad laws. Directed HoTT would allow you to guarantee that these laws are satisfied with much less work: you would only need a functor $M : \mathcal{U} \xrightarrow{+} \mathcal{U}$, natural transformations $\mathsf{return} : \mathrm{id} \rightsquigarrow M$ and $\mathsf{join} : (M \circ M) \rightsquigarrow M$ and a proof of associativity. Naturality would do the rest.

Moreover, isovariant dependent functions are constructive projective limits, which ought to have applications in computer science. For example, co-inductive types can be written as a projective limit. Isovariant dependent pairs are constructive injective limits, but as they don't seem to preserve decidability of equality (unless the collection of morphisms in the index category is manageable), they are probably only useful for enforcing contracts.

Denotational semantics often translate languages and theories to category theory. Although directed HoTT itself should not be trusted at this moment, it provides a nice combination of category and computer-friendliness, which is ideal for formal implementation of denotational semantics.

5.3 Discussion and further work

The theory as it stands, is quite a few loose ends. One way to make it more trustworthy, is by only allowing dependent pair and function types over type families of the form $a \overset{\times}{\mapsto} C(a, a)$ for some family $C : A \xrightarrow{-} A \xrightarrow{+} \mathcal{U}_k$, and by setting $= \circ \times \equiv \times$, which wouldn't change all that much to the results proven in this text. This would remove the need for bridges.

However, I think we should not just cut them out. Rather, I think one should try to understand them and reduce them to something that makes sense and looks good. Studying the behaviour of relations between types in HoTT may well be rewarding.

The consistency argument is exploratory at best. MLTT is likely not the best environment for interpreting directed HoTT. It would be good to find a more suitable environment, as better understanding of semantics may lead to answers about bridges and would allow a rule for creating e.g. covariant functions by proving covariance for an invariant function.

As the reader may have noticed, the one thing that makes directed HoTT interesting

for computer science – the fact that so much naturality work happens in the background – makes it a bit problematic for paper proving. Customizing a proof assistant to become compatible with this theory is probably necessary if further research *within* it is to be conducted.

Theorem 3.8.7 was a fascinating result. I think there is more to be found in that direction. For example, it is a known fact that inductive types are initial algebras, and co-inductive types are final algebras. Maybe we can actually state that and prove it in directed HoTT.

Finally, we didn't treat higher inductive types in the directed case at all. Nonetheless, they are a very interesting ingredient, especially for obtaining index categories to take limits over. There is no particular reason they were left out, apart from the fact that they did not appear in the basic results treated in chapter 3. An interesting question is whether we can view A as a higher inductive type over A^{core}. That would lead to a more general set of rules than just a rule that lets you prove a function's covariance.

Bibliography

[AMS07] Thorsten Altenkirch, Conor McBride, and Wouter Swierstra. Observational equality, now! In *Proceedings of the 2007 workshop on Programming languages meets program verification*, pages 57–68. ACM, 2007.

[BD09] Ana Bove and Peter Dybjer. Dependent types at work. In *Language engineering and rigorous software development*, pages 57–99. Springer, 2009.

[Coq92] Thierry Coquand. The paradox of trees in type theory. *BIT Numerical Mathematics*, 32(1):10–14, 1992.

[Dyb00] Peter Dybjer. A general formulation of simultaneous inductive-recursive definitions in type theory. *The Journal of Symbolic Logic*, 65(02):525–549, 2000.

[Gra09] Marco Grandis. *Directed Algebraic Topology: Models of non-reversible worlds*, volume 13. Cambridge University Press, 2009.

[Gro64] Alexandre Grothendieck. Catégories fibrées et descente. In *Revêtements étales et groupe fondamental*. Institut des Hautes Etudes Scientifiques, 1964.

[GZ67] Peter Gabriel and Michel Zisman. *Calculus of fractions and homotopy theory*, volume 35 of *Ergebnisse der Mathematik und ihrer Grenzgebiete*. Springer Berlin, 1967.

[Kun13] Kenneth Kunen. *Set Theory*, volume 34 of *Studies in Logic*. College Publications, 2013.

[LH11] Daniel R. Licata and Robert Harper. 2-dimensional directed type theory. *Electronic Notes in Theoretical Computer Science*, 276:263–289, 2011.

[Lic11] Daniel R Licata. *Dependently typed programming with domain-specific logics*. PhD thesis, Carnegie Mellon University, 2011.

[Lip11] Miran Lipovaca. *Learn You a Haskell for Great Good!: A Beginner's Guide*. No Starch Press, San Francisco, CA, USA, 1st edition, 2011. Consulted online version.

[ML84] Per Martin-Löf. Intuitionistic type theory. *Naples: Bibliopolis*, 1984.

[nLa12] nLab. Core. `http://ncatlab.org/nlab/show/core`, 2012. [Online; consulted version 6].

[nLa15] nLab. Grothendieck fibration. http://ncatlab.org/nlab/show/Grothendieck+fibration, 2015. [Online; consulted version 66].

[Nor09] Ulf Norell. Dependently typed programming in agda. In *Advanced Functional Programming*, pages 230–266. Springer, 2009.

[Shu11] Michael Shulman. Internal logic of a 2-category. http://ncatlab.org/michaelshulman/show/internal+logic+of+a+2-category, 2011. [Online; consulted version 15].

[Uni13] The Univalent Foundations Program. *Homotopy Type Theory: Univalent Foundations of Mathematics*. http://homotopytypetheory.org/book, Institute for Advanced Study, 2013.

[War13] Michael Warren. Directed type theory. https://video.ias.edu/univalent/1213/0410-MichaelWarren, 2013. Video lecture.

List of Symbols and Abbreviations

$\langle x \rangle$ A term x of MLTT, quoted as a term of directed HoTT. 142

brefl Constructor of the based identity type. 65, 68

$a \frown_A b$ Bridge type. 110

$\square^\frown$ Action of a function on bridges. 111

Cat Category of categories. 7

$\mathsf{catTransport}$ Categorical transport function. 90

$::$ Cons, the list or vector constructor that adds an element to the front. 34, 36, 59

A^{core} Core of A. 16, 47

cua Categorical univalence axiom. 135

$\square^\dagger$ Anti-involution on the bridge types. 110

‡ The empty list or vector. 34, 36, 59

$\mathsf{dirFunExt}$ Directed function extensionality. 129

$\mathsf{dirTransport}$ Directed transport function. 89

dua Directed univalence axiom. 132

$a =_A b$ Identity type; type of paths or proofs of equality. 64, 67

$=$ Isovariance. 15

$\square^=$ Action of a function on paths. 82

$!a =_A b$ Identity type based at a. 65, 68

$a \eqslantless_A b$ Type of pre-paths, equivalent to $\mathsf{gather}\, a =_{A^{\mathsf{loc}}} \mathsf{gather}\, b$. 141

$f \sim g$ Type of homotopies. 113

$\mathsf{Fin}(n)$ Finite type with n elements. 39, 40

m_n The mth element of $\mathsf{Fin}(n)$. 39

flip Constructor of A^{op}. 44

4 Variance annotation for functions from or to a bridgeless groupoid, when variance doesn't matter. 38

$\frac{c =_C c'}{p : a =_A b}$ Heterogeneous equality and morphism types. 94, 99, 100

$\frac{\mathsf{refl}\, c}{\mathsf{refl}\, a}$ Constructors of heterogeneous equality and morphism types. 94, 99, 100

$A \to B$ Type of non-dependent functions. 2, 21

$A \xrightarrow{v} B$ Type of non-dependent v-variant functions. 23

funExt Function extensionality. 28, 128, 129

gather Constructor of A^{loc}. 51

$f \succ g$ Type of natural transformations. 114

Grpd Category of groupoids. 9

happly Function that turns paths between functions into homotopies. 114, 116

HoTT Homotopy type theory. xi

id Constructor of the morphism type. 70

idbr Identity bridge. 110

ind_A Induction principle for the inductive type (family) A. 30

inl Left injection to the coproduct. 29, 31

inr Right injection to the coproduct. 29, 31

$[\![\mathcal{J}]\!]$ Interpretation of the judgement of directed HoTT $\mathcal{J}$ as a judgement in MLTT. 139

$[\![x]\!]_{\mathfrak{a}}$ Interpretation of the term of directed HoTT x as a term of type $\mathsf{El}\,\mathfrak{a}$ in MLTT. 140, 144

$[\![x]\!]$ Interpretation of the term of directed HoTT x as a term of type $\mathsf{El}\,\mathfrak{a}$ in MLTT, where we neglect to mention $\mathfrak{a}$. 140, 144

$\mathsf{isEquiv}(f)$ Proposition that f is an equivalence. 27, 86

$\mathsf{isIsom}(\varphi)$ Proposition that φ is an isomorphism. 89

$a \cong_A b$ Type of isomorphisms. 89

$\mathsf{isProp}(A)$ Proposition that A is a mere proposition. 88, 133

$\mathsf{isSet}(A)$ Proposition that A is a set. 133

$\mathsf{leftInv}(f)$ Type of left inverses of f. 86, 89

$\mathsf{leftTransport}$ Left transport function along a bridge. 110

$\lightning$ A symbol in the alphabet of MLTT that never occurs in true judgements. 140

$\mathsf{List}\,A$ Type of lists. 34, 36

A^{loc} Localization of A. 16, 51

m Assumed height of the universe tower in directed HoTT. 140

$(x : A) \mapsto b[x]$ Function that maps $x : A$ to $b[x]$. 22, 25

$(x : A) \overset{v}{\mapsto} b[x]$ The v-variant function that maps $x : A$ to $b[x]$. 23, 26

$-$ Contravariance. 15

MLTT Martin-Löf type theory. xi

$a \rightsquigarrow_A b$ Type of morphisms. 70

$\square^{\rightsquigarrow}$ Action of a function on morphisms. 84

$!a \rightsquigarrow_A b$ Morphism type based at a. 71

$\mathbb{N}$ Type of natural numbers. 36, 37

napply Function that turns morphisms between functions into natural transformations. 116

1 Unit type. 2, 39, 40

A^{op} Opposite of A. 16, 44

$(\square, \square)$ Pair constructor in symmetric HoTT; non-dependent or covariant dependent pair constructor in directed HoTT. 41, 42, 53, 56

$(\square \overset{v}{,} \square)$ The v-variant dependent pair constructor. 55

$A + B$ Coproduct/disjoint union. 2, 29, 31

$+$ Covariance. 15

prerefl Constructor of the pre-path type. 141

prl Left projection from the product. 41, 43, 54, 56, 57

$\prod_{x:A} C(x)$ Type of dependent functions. 3, 25

$\prod^{v}_{x:A} C(x)$ Type of v-variant dependent functions. 26

prr Right projection from the product. 41, 43, 54, 56

qInv(f) Type of quasi-inverses of f. 86

rec_A Recursion principle for the inductive type (family) A. 29

refl Constructor of the identity type. 64, 67

$\mathsf{Rel}_C(\beta)$ Relation along a bridge. 111

rightInv(f) Type of right inverses of f. 86, 89

rightTransport Right transport function along a bridge. 110

sbid Source based morphism constructor. 71

$A \simeq B$ Type of equivalences. 27

$A \overset{+}{\simeq} B$ Type of (covariant) equivalences. 28, 86

$\star$ The element of the unit type. 2, 39

strip Constructor of A^{core}. 47

succ n Successor of the natural number n. 37

$\sum_{x:A} C(x)$ Type of dependent pairs. 3, 53

$\sum^{v}_{x:A} C(x)$ Type of v-variant dependent pairs. 55

tbid Target based morphism constructor. 71

3 Variance annotation for functions from a bridged groupoid, where $+$, - and $\times$ are interchangeable. 50

$\times$ Invariance. 15

$A \times B$ Cartesian product. 2, 41, 42

toBrid Function that weakens morphisms to bridges. 110

toMorph Function that weakens paths to morphisms. 82

transport Transport function. 88

2DTT 2-dimensional directed type theory. 6

ua Univalence axiom. 131

uip Uniqueness of identity proofs. 5

unflip Eliminator of A^{op}. 46

$\mathcal{U}_k$ The kth universe. 20

unstrip Eliminator of A^{core}. 48

$\mathsf{Vec}_n\, A$ Type of vectors of length n. 59

0 Empty type. 2, 38, 39

zero Zero, the natural number. 37

Index

∞-groupoid, xi, **5**, 78

Agda, 4
axiom K, 5

bivariance, 108
bridges, xii, 15, **105**

category, 134
coherence laws, 79
composition
    of functions, **22**, **24**
    of variance, 16
computation rule, **22**
conjunction, 2
constancy, 106
context, **14**
contravariance, 10, **15**
coproduct, 2, **28**, 119
core, 7, 9, 16, **47**, 123
covariance, 10, **15**
cumulativity of universes, 20
Curry-Howard correspondence, 2
currying
    core, 48
    dependent, 54, 55
    localization, 51
    non-dependent, 41, 43
    opposite, 45
    variance, 45, 48, 51

dimension, 6
directed function extensionality, **129**
disjunction, 2

element relation, 3
empty type, 38
equality
    judgemental, xi, 3, **14**
    propositional, xi, 3, **64**
equivalence, 27, **86**
    logical, 2

false, 2
family of finite sets, 39
fibration, 92
first order logic, 2

formation rule, **21**
function
    dependent, 3, **25**, 128
    extensionality, 28
    non-dependent, **21**
function extensionality, **128, 129**

Grothendieck fibration, 97
groupoid
    bridged, 49, 116
    bridgeless, 52, 118

higher inductive type, 75
homotopy, 113
homotopy hypothesis, xi, 5

identity
    function, **22**, **24**
    isovariant, 38, 40, 49, **52**, 52
    morphism, 70
identity type, 3, **64**
implication, 2
induction principle, 30
inductive type, 28
    family, 57
inhabited, 1
injective limit, 126
introduction rule, **22**
invariance, xii, 10, **15**
inverse
    left, 86
    quasi-, 86
    right, 86
isomorphism, 89
    in categories in symmetric HoTT, 134
isovariance, xii, 10, **15**

judgement, 1, **13**

law of excluded middle, 2
localization, 9, 16, **51**, 123, 141

Martin-Löf type theory, xi
mere propoisition, 133
mere proposition, 88
metatheory, 139

natural equivalence, 113
natural numbers, 36, 120
natural transformation, 114
negation, 2

opposite, 9, 16, **44**, 122

pair
    dependent, 3, **53**, 125
path, **64**
path induction, 65
    based, 65
pre-paths, 141
precategory, 133
product, 2, **41**, 125
projective limit, 129

quantification
    existential, 3
    universal, 2

recursion principle, 29

set, 133
symmetric HoTT, **xii**

total space, 92, 97
transport, 88
    categorical, 90
    directed, 89
    in categories in symmetric HoTT, 134
true, 2
type family, 57

uniqueness of identity proofs, xi, 3, **5**
uniqueness principle, **22**
unit type, 39
univalence, xi, 9, **131**
    categorical, 135
    directed, 132
    theorem in directed HoTT, 137
univalence axiom, 5
universe, **20**, 130

variance
    higher order, 10
vector, 59

W-types, 74

zigzag, xii, 106, 109, 141

iMinds-DistriNet Research Group
Celestijnenlaan 200A
3001 Heverlee, BELGIË
tel. + 32 16 32 76 40
fax + 32 16 32 79 96
www.kuleuven.be