Univalent Types, Sets and Multisets

Investigations in dependent type theory

Håkon Gylterud

Academic dissertation for the Degree of Doctor of Philosophy in Mathematics at Stockholm University to be publicly defended on Thursday 9 February 2017 at 13.00 in sal 14, hus 5, Kräftriket, Roslagsvägen 101.

Abstract

This thesis consists of four papers on type theory and a formalisation of certain results from the two first papers in the Agda language. We cover topics such as models of multisets and sets in Homotopy Type Theory, and explore ideas of using type theory as a language for databases and different ways of expressing dependencies between terms. The two first papers build on work by Aczel 1978. We establish that the underlying type of Aczel's model of set theory in type theory can be seen as a type of multisets from the perspective of Homotopy Type Theory, and we identify a suitable subtype which becomes a model of set theory in which equality of sets is the identity type. The third paper is joint work with Henrik Forssell and David I .Spivak and explores a certain model of type theory, consisting of simplicial complexes, from the perspective of database theory. In the fourth and final paper, we consider two approaches to unraveling the dependency structures between terms in dependent type theory, and formulate a few conjectures about how these two approaches relate.

Keywords: *type theory, homotopy type theory, dependent types, constructive set theory, databases, formalisation, agda.*

Stockholm 2017
http://urn.kb.se/resolve?urn=urn:nbn:se:su:diva-136896

ISBN 978-91-7649-654-1
ISBN 978-91-7649-655-8

Department of Mathematics

Stockholm University, 106 91 Stockholm

Univalent Types, Sets and Multisets
— Investigations in dependent type theory

Håkon Robbestad Gylterud

Univalent Types, Sets and Multisets

Investigations in dependent type theory

Håkon Robbestad Gylterud

ISBN 978–91–7649–654–1
ISBN 978–91–7649–655–8 (PDF version)

Printed in Sweden by US-AB, Stockholm 2017

Distributor: Department of Mathematics, Stockholm University

Sammendrag

Avhandlingen består av fire artikler i matematisk logikk og én formaliseringdel. De fire artiklene er arbeider innenfor området typeteori. De to første artiklene er arbeider internt i typeteorien, og formaliseringen av disse er skrevet i Agda – et bevissjekkingssystem basert på Martin-Löfs type teori.

Den første artikkelen omhandler *multimengder* i typeteori. Multimengder er et kjent begrep fra områder som kombinotarikk og informatikk. Kort beskrevet er multimengder samlinger av elementer hvor et element kan forekomme et vilkårlig antall ganger i samlingen. Formålet med artikkelen er å beskrive et hierarki av iterative multimengder, og utforske aksiomer for disse som ligner de man kjenner fra konstruktiv mengdelære. Homotopitypeteori og Voevodskys univalensaxiom spiller en sentral rolle, ettersom hierarkiet av multimengder bygges relativt til et univalent univers.

Den andre artikkelen tar ibruk hierarkiet av iterative multimengder fra den første artikkelen, og utvikler en modell for mengdelæren i homotopitypeteori. Dette gjøres ved å definiere mengder som de multimengder hvor hvert element forekommer høyst én gang. Vi viser at denne modellen tilfredstiller aksiomer for konstruktiv mengdelære, og at den er ekvivalent til en allerede kjent modell for mengdelæren. En av attraksjonene ved denne formuleringen er at den kan uttrykkes uten såkalte *høyere induktive typer*.

De to artikklene om multimengder og mengder er formalisert i Agda, og bevisene er sjekket ved hjelp av datamaskin. Kildekoden for formaliseringen er gjengitt, sammen med en kort diskusjon, som en separat del i denne avhandlignen.

De to siste artikklene omhandler semantikk for typeteori fra et kategoriteoretisk perspektiv. Først diskuteres en mulig kobling mellom typeteori og databaseteori, ved å konstruere en modell for typeteorien basert på simplisialkomplekser. Vi utforsker hvordan ulike typeteoretiske konsepter, slik som Σ-typer, Π-typer og univers kan oversettes via denne modellen til databaseteoretiske termer. For eksempel viser det seg at *naturlig join* er et spesialtilfelle av Π-typen i denne modellen.

Den siste artikkelen diskuterer to ulike måter å beskrive avhengighetsrelasjoner mellom termer i avhengige typesystemer. Den første måten er en variant av *kategorier med attributter*, som er en vellstudert måte å gi kategoriteoretisk semantikk til typeteori. Den andre er en videreutvikling av Makkai's såkalte enveiskategorier, til å inkludere termer i tillegg til typer.

Contents

Sammendrag v
Preface xi
Introduction to the thesis xv

A Multisets in Type Theory **23**
1 Introduction 25
2 Notation and background 27
3 The model 28
3.1 Aczel's model 28
3.2 The identity type on W-types 30
3.3 A model of multisets 30
3.4 Equality and the identity type 31
3.5 Extensionality 32
4 Multiset constructions 33
4.1 Restricted separation 34
4.2 Union-replacement 35
4.3 Singletons 36
4.4 Pairing 37
4.5 Ordered Pairs 39
4.6 Cartesian products 39
4.7 Functions 40
4.8 Fullness, subset collection and operations 41
4.9 Exponentiation 42
4.10 Natural numbers 44
5 Homotopic aspects of M 45
5.1 Homotopy n-type 45
5.2 HITs and multisets 46

B From multisets to sets in Homotopy Type Theory 49
1 Introduction . . . 51
1.1 From multisets to sets – two ways . . . 52
1.2 Outline . . . 53
1.3 Notation . . . 53
2 Types and propositions . . . 54
3 Models where equality is identity . . . 55
3.1 $\in$-structures . . . 56
3.2 Translations of first-order logic into type theory . . . 56
3.3 Axioms of set theory . . . 58
4 The model of iterative sets . . . 59
5 Basic results . . . 61
6 V models Myhill's constructive set theory . . . 62
6.1 Extensionality . . . 62
6.2 The empty set, natural numbers . . . 62
6.3 Separation . . . 63
6.4 Pairing and Union . . . 63
6.5 Replacement . . . 64
6.6 Exponentials . . . 65
7 Collection axioms . . . 66
7.1 Strong Collection . . . 67
7.2 Subset Collection . . . 68
8 Equivalence with the HIT-formulation . . . 70

C Agda Formalisation 73
1 Prerequisites . . . 75
2 Assumptions of the formalisation . . . 76
3 Case study . . . 77
4 Selected excerps from HoTT-agda . . . 79
5 Code . . . 80
5.1 Propositions/Equivalences.magda . . . 80
5.2 Propositions/Existensial.magda . . . 82
5.3 Propositions/Disjunction.magda . . . 82
5.4 W/W.magda . . . 83
5.5 Function/Fiberwise.magda . . . 85
5.6 Multiset/Iterative.magda . . . 89
5.7 Sets/Iterative.magda . . . 91
5.8 Sets/Axioms.magda . . . 96

D Type Theoretical Databases **103**
1 Introduction . . . 105
2 The model . . . 108
2.1 Complexes, schemas, and instances . . . 108
2.2 Structure of the model . . . 114
3 The type theory . . . 117
3.1 The type theory $\mathcal{T}$ . . . 118
3.2 Instance specification as type introduction . . . 118
4 Universe . . . 120
4.1 Constructing the universe . . . 120
4.2 Using the universe in the type theory . . . 123
5 Representing data simplicially . . . 124

E Dependent Term Systems **135**
1 Introduction . . . 137
2 Notation . . . 138
3 Categories with attributes and definitions . . . 139
3.1 Categories with Attributes . . . 140
3.2 Two simple examples . . . 141
3.3 Categories with definitions . . . 142
3.4 Comparison with categories with families . . . 147
3.5 Morphisms . . . 148
3.6 Free essentially CwAs . . . 149
3.7 Analogy with sharing . . . 153
4 One-way Categories . . . 154
4.1 Structures . . . 154
4.2 Extending structures . . . 155
4.3 The category with attributes of structures . . . 157
4.4 The category of corollas . . . 158
5 One-way Term Systems . . . 160
5.1 Structures . . . 165
5.2 Structure homomorphisms . . . 168
5.3 Connecting term systems with CwDs . . . 168
6 Future work . . . 170
6.1 Rules which introduce equalities . . . 171
6.2 A weaker notion of homomorphism . . . 171
6.3 Working out 2-categorical properties of CwDs . . . 171

Preface

This text has been written as a doctorate thesis in the subject of mathematical logic, and is a collection of papers. The thesis is based on research carried out during the years 2012–2016, and consists of five parts. The parts A, B, D and E are individual research papers, while Part C is a formalisation of Part A and Part B in the Agda language. Each part is equipped with an abstract and more careful introduction. We will here give a bit of context for each part.

Part A. Having studied containers for my Master Thesis at The University of Oslo, 2011, it was natural to continue to study polynomial functors and W-types in the context of Martin-Löf type theory when I arrived in Stockholm, January 2012. Through reading Egbert Rijke's master thesis, and attending the 4th Formal Topology Workshop in Ljubljana, June 2012, I became aware of what is now called Homotopy Type Theory, and the novel interpretation of the identity type as paths in a space.

In late 2013 I was studying the W-types of groupoids and their identity type when I considered the W-type, $W_{a:U} T\, a$ for a universe T. Erik Palmgren, my advisor, quickly pointed me to Aczel's 1977 paper, which uses this exact type to model set theory. Applying what homotopy theory tells us about the identity type of the universe, I arrived at the conclusions found in Part A.

Part B. The work on constructing a model of constructive set theory from the multisets of Part A started while I was visiting Carnegie Mellon University in Pittsburgh, Pennsylvania, late January and early February of 2014. At the seminar there, I presented my ideas, and Steve Awodey raised the question of how to turn this into a model of set theory. Answering this question then became the focus of Part B of this thesis.

Part C. While in Pittsburgh, I also started formalising my results on multisets in Agda. Having experimented with Agda since the very first weeks of coming to Stockholm, I was happy to find that my work on multisets was very amendable to formalisation. The work on formalising these results continued for more than a year, coming to essential completion in August 2015, after a quiet month of focused effort in the pleasant Stockholm summer. It is now collected in Part C.

Part D. The fourth part is based on previous work by co-author David I. Spivak on the connections between simplicial complexes and databases. Along with Henrik Forssell, who initiated the cooperation, we worked out the details of a model of type theory based on simplicial

complexes (which form a locally cartesian closed category), and made connections back to notions in databases such as natural join. In January 2016 I presented this at Logical Foundations of Computer Science (LFCS16), and a shorter version of the article was printed in the proceedings of the conference. The full version of Part D is submitted for the post-conference special volume of Annals of Pure and Applied Logic.

Part E. During the spring term 2013 I took a course on the Theory of Operads, taught by Sergei Merkulov. Inspired by the operad approach to algebras, and Makkai's one-way categories, I wanted to study dependent type theory from a more combinatorial perspective. Some small progress on this topic was made in the following couple of years and presented in various forms at the Stockholm Logic Seminar. This work has now been collected into Part E.

Organisation of the thesis

The thesis is divided into five parts, referred to by the Latin letters A, B, C, D and E. Each part contains a number of sections, numbered 1, 2 etc. The sections are sometimes subdivided into subsections: 3.1, 3.2, etc. Definitions, lemmas, propositions and theorems are collectively numbered within each part. For instance, Lemma A:6 is followed by Definition A:7. The parts each start with an abstract and the first section of each part is an introduction. The list of references are found at the end of each part.

The mathematical notation varies slightly between the parts, reflecting that these are individual works of mathematics here collected. Hopefully, the reader will find that each part introduces its notation clearly.

Acknowledgements

The list is long of people whose discussions and encouragements have helped form and motivate the work presented here. Special thanks go to my advisor Erik Palmgren, who patiently has supported my work, and contributed his immense experience and knowledge.

Henrik Forssell and David I. Spivak, my coauthors on Part D, I would like to thank for their cooperation. Henrik has furthermore been the coadvisor of my thesis work. I would like to thank him for including me and inviting me to cooperate with him on several research projects, and for his office door always being open when I have needed someone to discuss with.

The logic group at the Mathematical Department of Stockholm University has been a really great environment to do research in. Erik and

Henrik I have already mentioned. Many thanks to them and the rest of the group: Per, Peter, Christian, Jacopo, Johan and Anna — you have been great colleagues and your own research has greatly inspired mine.

During my time at Stockholm University, I have also been fortunate enough to visit many other institutions in Europe and North America. Thanks go to all the wonderful researchers I have had the honour to meet, for their questions, comments and discussions — in particular to Steve Awodey of Carnegie Mellon University whose questions inspired what would become Part B of this thesis.

Last, but not least, I would like to thank my family for their support and help — Inna for her loving support, Yngvar for his kindness and good mood, Ingrid for long talks and proof-reading and my parents for always being there for me and from an early age allowing me to pursue my interests in mathematics. No research could have been done without such a great home support team.

—Håkon Robbestad Gylterud
Stockholm, 2017

Introduction to the thesis

Each part of the thesis has an individual section devoted to introduction. This part is intended as a quick introduction, focusing on the ideas behind each part.

Type theory

Martin-Löf's intuitionistic type theory serves as foundation of constructive mathematics. For a complete introduction we refer the reader to Nordström, Petersson, and Smith 2000. We will here give a high level overview of the aspects relevant to the articles of this thesis.

At the core of Martin-Löf's type theory are ideas such as

- *propositions as types*, sometimes referred to as the Curry—Howard correspondence,
- defining inductive structures through *introduction and elimination rules*, and
- collecting types into *universes*, in a way similar to Grothendieck universes do in set theory.

The way these three are accomplished is by having *dependent types*. A dependent type is a type which takes parameters in other (possibly themselves dependent) types. A typical first example is the type of vectors over some base type, say A. The vectors have different length and thus one can see vectors as a type $\mathrm{Vec}_A\ n$ in the context $(n : \mathbb{N})$, meaning that n for each natural number n there is a type $\mathrm{Vec}_A\ n$ of vectors of length n, of elements of A. One sometimes use the term "family of types" to denote dependent types.

Often, one can express dependent types as functions into a type of types. For instance, if Type is the type of small types, then a family of small types parameterised by a type A can be represented by a function $A \to \mathrm{Type}$.

Propositions as types

Perhaps the most appealing aspect of dependent type theory is that one does not need a separate framework for logic. Instead the type theory comes equipped with a logical framework — where types play the role of propositions, and in particular dependent types play the role of predicates. For instance, a unary predicate on a type A is simply a dependent type $P : A \to \mathrm{Type}$. A binary predicate can be seen either as

a dependent type $A \times A \to \text{Type}$ or, more conveniently, $A \to A \to \text{Type}$ — by currying.

The beauty of representing propositions by types is that it turns out that the logical connectives can be expressed by the usual type formations, such as dependent products and sums. For instance the existential quantification of a binary predicate $P : A \to \text{Type}$ is expressed by $\sum_{a:A} P\, a$. The elements of $\sum_{a:A} P\, a$ are pairs (a, p) where $a : A$ and $p : P\, a$, which is exactly a witness of the existential quantification there is $a : A$ such that $P\, a$ holds (i.e. has a witness $p : P\, a$). The table below summarises the correspondence.

Logical connective	Type formation
$\forall a : A\, P\, a$	$\prod_{a:A} P\, a$
$\exists a : A\, P\, a$	$\sum_{a:A} P\, a$
$P \to Q$	$P \to Q \equiv \prod_{p:P} Q$
$P \wedge Q$	$P \times Q \equiv \sum_{p:P} Q$
$P \vee Q$	$P + Q$
$\bot$	0, the empty type
$\top$	1, the unit type

Since a type may have more than one element, logic as presented above is called *proof-relevant* logic. The idea is that each element of the type representing a proposition represents a proof of that proposition. An added benefit of having an element of a type representing a proof of a proposition is that the element may be normalised — an important property of type theory. This means that one can often extract algorithms from proofs in type theory.

Proof-relevance is especially interesting in the case of equality. There are several ways to represent equality in type theory. One way, rooted in the ideas of Errett Bishop, is to equip the type with a separate equality predicate to form what is called a *setoid*. A second way is to represent equality by what is called *the identity type*. Given any type A, the identity type, Id_A is inductively defined as the least reflexive relation on the type.

An amazing fact, first established by Hofmann and Streicher[1] by their groupoid interpretation of type theory, is that the identity type may have more than one element. Thus, two elements of a type may be equal in more than one way. This has become the foundation of what is called *Homotopy Type Theory* — where types are interpreted as

[1]Hofmann and Streicher 1998.

a space and the identity type is the path space. For a comprehensive introduction to this field, see the book "Homotopy Type Theory"[2].

One of the deeper notions which has been brought to attention by Homotopy Type Theory is the notion of equivalence of types. We refer to the book, "Homotopy Type Theory", for the definition, but we will use the notation $A \simeq B$ to denote that A and B are equivalent types.

Introduction and elimination rules

In the previous subsection we mentioned different type constructors, such as Π-types and Σ-types. In Martin-Löf's type theory these are primitive operations on types, presented each by a set of rules. For each type there is a formation rule, none or more introduction rules, an elimination rule, and none or more computation rules. We will not display these rules here, but rather give some intuition as to what they express.

In short, the formation rules tell us how to construct types, and introduction rules how to construct elements of types. Elimination rules give sufficient conditions to carry out a construction with a free variable in the type. For instance, the elimination rule of $\mathbb{N}$ corresponds to mathematical induction by propositions as types. Computation rules tell us the result of applying the a construction specified by an elimination rule to an element constructed by an introduction rule. We refer the reader to Nordström, Petersson, and Smith 2000 for a complete description of these concepts.

Universes

In the usual formulations of dependent type theory there is one kind of types which does not have an elimination — namely the universes. The intuition behind universes is that they are "open-ended" families of types, closed under type formation rules such as Π-types and Σ-types. This means that if $A : U$ is a type in the universe[3] and $F : A \to U$ is a family of types in U, indexed by A, then $\sum_{a:A} F\, a : U$, etc.

The lack of an elimination rule has the consequence that the identity type on U is undecided. This leaves room for additional axioms specifying how to interpret the identity of the universe. The most famous

[2]Univalent Foundations Program 2013.

[3]Elements of types such as U are not themselves types, and one often speak of a decoding family $T : U \to \text{Type}$, when defining a universe formally. For simplicity, we apply the syntactic convention which omits mention of this decoding family, and write $A : \mathit{Type}$ instead of $T\, A : \text{Type}$

such axiom is *Voevodsky's Univalence Axiom*. It states that the identity type on the universe coincides with equivalence of types. Concretely, it states that for each $A, B : U$ the canonical map $Id_U\, A\, B \to A \simeq B$, is an equivalence of types.

Multisets

Multisets can be simply described as collections of elements where each element may occur any number of times. Examples, such as $\{1, 1, -2\}$, are abundant in mathematics, for instance as the roots of polynomials, such as $x^3 - 3x + 2$. One can even view polynomials with coefficients in natural numbers as finite multisets of finite multisets of variables. For instance $xy^2 + 3xy + x + 2$ could be represented by $\{\{x, y, y\}, \{x, y\}, \{x, y\}, \{x, y\}, \{x\}, \{\}, \{\}\}$.

Going beyond the finite case, any function gives rise to a *multiset image*, where each element in the codomain occurs the number of times the function attains this value. For instance $f : \mathbb{R} \to \mathbb{R}$, given by $f\, x := x^3 - 3x + 2$ (see Figure 1), would have an image multiset: Im $f = (-\infty, 4) \cup [0, 4] \cup (0, \infty)$. Union of multisets is additive, so that the number of times x occurs in $A \cup B$ is the sum of the times x occurs in A and the times x occurs in B. Thus, Im f is the multiset in which each element of $(0, 4)$ occurs trice, 4 and 0 occur twice, and elements of $(-\infty, 0)$ and $(4, \infty)$ occur once. This gives a lot more information about the polynomial, compared to *the image set* — which is just $\mathbb{R}$ for any polynomial of degree 3.

A multiset may also be infinite in the sense that an element may occur infinitely main times. For instance 0 occurs countably infinitely many times in the image of the sine function, since $\sin x = 0 \;\Leftrightarrow\; \exists k \in \mathbb{Z}\; x = \pi k$.

The first article of this thesis concerns multisets. In particular *iterative multisets*. Quoting from the introduction of Part A:

> In a flat multiset, the elements are taken from some domain which may not consist of other multisets. The iterative multisets have elements which are multisets themselves, and the collection iterative multisets is generated in a well-founded manner.

The idea is to have a similar structure as in usual (iterative) set theory, where there is a domain V of sets and a binary relation $\in$ on V. This is where the idea of propositions as types enter the picture. We will have a domain M and a binary relation $\in$ on M. However, since we

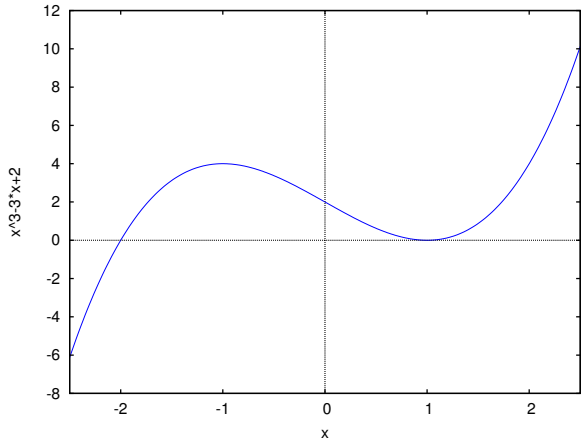


Figure 1: Plot of $f\,x := x^3 - 3x + 2$.

will be working in type theory, $x \in y$ will be a *type* for all $x, y : M$. The natural interpretation of the elements of $x \in y$ is that they represent the occurrences of x in y, and thus proof-relevance dictates that x may occur multiple times in y, hence they are multisets. We see that merely stating the signature of set theory in type theory has brought us to consider the possibility of multisets. Thus, the type M will be the type of iterative multisets M.

Letting $x \in y$ be a type is a convenient notation for multisets. For instance, stating that 0 occurs countably many times in $\operatorname{Im} \sin$ is simply asserting that $(0 \in \operatorname{Im} \sin) \simeq \mathbb{N}$.

We have already mentioned that any function gives rise to a multiset. Part A, is based on the idea that this is an adequate way to represent multisets in general, and in particular that an iterative multiset can be seen as a type $A : U$ and a function $f : A \to M$. This gives rise to an inductive definition, namely that M is the least solution to the equation $M \simeq \sum_{A:U} (A \to M)$. This is an example of a well-known inductive construction, namely W-types. In fact this is the exact type studied by Aczel 1978, in his construction of a model of set theory in type theory.

In his work, Aczel uses the setoid approach to equality. The Univalence Axiom, however, allows us to compute the identity on M. Interestingly, the identity type on M is non-trivial, with several distinct equalities even between concrete finite multisets in M. Thus, M is a groupoid, and we can see multiset theory as a kind of categorification of set theory.

Sets

The idea behind Part B is that iterative sets are merely a special class of iterative multisets, namely those in which each element occurs at most once — and such that this property is hereditary, so that each element again has the iterative set property. We define such a subtype of M, by induction, and consider how various axioms of constructive set theory apply to this model. Quoting from the introduction:

> Once the notion of a multiset is defined, it is natural to study the hereditary subtype of multisets where each element occurs at most once. These are in a certain sense the most natural representations of iterative sets from a homotopy type theory point of view. These are namely the multisets for which the elementhood relation is hereditarily, merely propositional (type level -1).
>
> In this text we explore how this type models various axioms of constructive set theory. We also show that it is equivalent to the higher inductive type outlined in the book "Homotopy Type Theory"[4].

Databases

While the first two articles of the thesis are completely situated inside type theory, Part D takes a step out and considers particular a model of type theory from a category theoretic point of view. The particular model is based on simplicial complexes, and is intended to model certain aspects of database theory. Quoting from the introduction:

> Databases being, essentially, collections of (possibly interrelated) tables of data, a foundational question is how to best represent such collections of tables mathematically in order to study their properties and ways of manipulating them. The relational model, essentially treating tables as structures of first-order relational signatures, is a simple and powerful representation. Nevertheless, areas exist in which the relational model is less adequate than in others. One familiar example is the question of how to represent partially filled out rows or missing information. Another, more fundamental perhaps, is how to relate instances of different schemas, as

[4]Univalent Foundations Program 2013.

opposed to the relatively well understood relations between instances of the same schema. Adding to this, an increasing need to improve the ability to relate and map data structured in different ways suggests looking for alternative and supplemental ways of modelling tables, more suitable to "dynamic" settings. It seems natural, in that case, to try to model tables of different shapes as living in a single mathematical structure, facilitating their manipulation across different schemas.

We investigate, here, a novel way of representing data structured in systems of tables which is based on simplicial sets and type theory rather than sets of relations and first-order logic.

The basic notions of databases are those of a database *schema* and those of an *instance of a schema*. Simply put, a schema is a description of a layout of tables: each table described by a list of *attributes*. It is essential that attributes may be shared across different tables. An instance is then an actual set of tables, filled with data, which adhere to the layout of the schema.

Given a schema and an instance, *full tuple* is a tuple of data with an entry for each attribute in the schema, such that the restriction to each table corresponds to an existing row in the instance. Below is a simple example.

Schema: $\{(A, B, C), (A, D)\}$
Instance:

A	B	C
x	a	3
y	b	7
x	b	1

A	D
x	$\top$
x	$\bot$
y	$\bot$
z	$\top$

Full tuples: $(x, a, 3, \top)$, $(x, c, 1, \top)$, $(x, a, 3, \bot)$, $(x, c, 1, \top)$ and $(y, b, 7, \bot)$.

The idea we investigate in Part D is to align these three basic notions, along with a further notion of morphism between schemas, with the basic judgements of type theory. The following table, from the article summarises this alignment:

Judgement	Interpretation
$\Gamma : \mathtt{Context}$	$[\![\Gamma]\!]$ is a schema
$A : \mathtt{Type}(\Gamma)$	$[\![A]\!]$ is an instance of the schema Γ
$t : \mathtt{Elem}(A)$	$[\![t]\!]$ is an full tuple in the instance A
$\sigma : \Gamma \longrightarrow\!\!\!\!\!\rightarrow \Lambda$	$[\![\sigma]\!]$ is a (display) schema morphism
$\Gamma \equiv \Lambda$	$[\![\Gamma]\!]$ and $[\![\Lambda]\!]$ are equal schemas
$A \equiv B : \mathtt{Type}(\Gamma)$	$[\![A]\!]$ and $[\![B]\!]$ are equal instances of $[\![\Gamma]\!]$
$t \equiv u : \mathtt{Elem}(A)$	$[\![t]\!]$ and $[\![u]\!]$ are equal full tuples in $[\![A]\!]$
$\sigma \equiv \tau : \Gamma \longrightarrow\!\!\!\!\!\rightarrow \Lambda$	the morphisms $[\![\sigma]\!]$ and $[\![\tau]\!]$ are equal

References

Aczel, Peter (1978). "The Type Theoretic Interpretation of Constructive Set Theory". In: *Logic Colloquium '77*. Ed. by A. MacIntyre, L. Pacholski, and J. Paris. North–Holland, Amsterdam-New York, pp. 55–66.

Hofmann, Martin and Thomas Streicher (1998). "The groupoid interpretation of type theory". In: *Twenty-five years of constructive type theory (Venice, 1995)*. Vol. 36. Oxford Logic Guides. New York: Oxford Univ. Press, pp. 83–111.

Nordström, B., K. Petersson, and J. M. Smith (2000). *Martin-Löf's Type Theory*.

Univalent Foundations Program, The (2013). *Homotopy Type Theory: Univalent Foundations of Mathematics*. Institute for Advanced Study: homotopytypetheory.org. URL: `http://homotopytypetheory.org/book`.

A

Multisets in Type Theory

Abstract

A multiset consists of elements, but the notion of a multiset is distinguished from that of a set by carrying information of how many times each element occurs in a given multiset. In this work we will investigate the notion of iterative multisets, where multisets are iteratively built up from other multisets, in the context Martin-Löf Type Theory, in the presence of Voevodsky's Univalence Axiom.

Aczel 1978 introduced a model of constructive set theory in type theory, using a W-type quantifying over a universe, and an inductively defined equivalence relation on it. Our investigation takes this W-type and instead considers the identity type on it, which can be computed from the Univalence Axiom. Our thesis is that this gives a model of multisets. In order to demonstrate this, we adapt axioms of constructive set theory to multisets, and show that they hold for our model.

1 Introduction

The purpose of this paper is to describe a model of iterative, transfinite multisets and to discuss a possible axiomatisation of the model in the context of univalent Martin-Löf style type theory. Before describing the model, we discuss existing works on multisets and their relation to the model at hand.

Usage of multisets has a long history, both in mathematics and in applications. In classical mathematics one models multisets inside set theory in various ways. Here follows a brief description of three common ways of representing multisets.

A very general definition, introduced in Rado 1975, is that a multiset on a domain set X, consists of an assignment $X \to \mathrm{Card}$, which for each element of the domain specifies the (possibly transfinite) number of occurrences of the element in the multiset. Often, this notion is restricted to functions $X \to \mathbb{N}$, which represent multisets where each element occurs finitely many times.

One can also view a multiset as a set A with an equivalence relation R defined on A. The idea is that the elements of A are the occurrences

in the multiset, and the relation R specifies which occurrences are the same. Thus the number of occurrences of $a \in A$ is the size of the R-equivalence class of a.

A third way is to consider a multiset as a family of sets. The index set of the family corresponds to the domain in Rado's multisets, but instead of assigning a cardinal number, we have a set of occurrences.

These three approaches illuminate different aspects of multisets, and even though they are formulated quite differently it is relatively easy to pass back and forth between them. In fact they would be equivalent if one removes the constraint that the relation in the second formulation should be reflexive, or restricts the other two to ensure that each element in the domain occurs at least once.

Rado's formulation reflects that elements in a multiset occurs a specific number of times. This can be problematic in a constructive context, where the notion of cardinality is much more nuanced. This is solved if one takes the family-of-sets definition, which makes perfect sense constructively, but requires more thought as to what constitutes equality between multisets. The notion of equality between multisets is a topic we will come back to later in this paper.

Considering a multiset as a set with equivalence relation, a setoid, gives an interesting way to talk about the different between identical elements and equal elements. The identity of elements in the underlying set A tells us when occurrences are identical, and the relation R tells us which occurrences are equal. Since the underlying theory is set theory, we can distinguish equal occurrences in a multiset, but not identical occurrences.

All three notions describe what we in this paper will refer to as "flat multisets", as opposed to "iterative multisets". In a flat multiset, the elements are taken from some domain which may not consist of other multisets. The iterative multisets have elements which are multisets themselves, and the collection iterative multisets is generated in a well-founded manner.

Blizard 1988 develops an axiomatisation of iterative multisets with finite occurrences. The theory is a two-sorted, first-order theory. The two sorts N and M, represent the natural numbers and multisets respectively. The natural numbers are given by the Peano axioms. Membership is interpreted as a ternary predicate $- \in_{-} -$, where the intended interpretation of $x \in_n y$ is that x occurs exactly n times in y. The axioms for multisets are then chosen so that one can reconstruct ZFC internally as the multisets where each element occurs at most once.

In this paper, we will take a different view on elementhood of multi-

sets compared to Blizard. Instead of a ternary relation, we will keep the $\in$-relation binary and invoke the propositions-as-sets attitude of Martin-Löf type theory.

In Martin-Löf type theory one does not generally distinguish the notion of set from the notion of proposition. The notion of a set, as given by its canonical elements, corresponds exactly to the notion of a proposition as given by its canonical proofs. This leads us to give the binary relation $\in$ the typing $\in : M \to M \to \text{Type}$. Thus, for given $x, y : M$, we have that $x \in y$ is a set. *The natural interpretation is that $x \in y$ is the set of occurrences of x in y.*

Taking the idea of using types to capture the number of occurrences further, we need a notion of equivalence of type. This is where Univalent Type Theory enters the picture. Voevodsky's univalence axiom expresses[5] that an identity between type is exactly (equivalent to) an equivalence. For types which are mere sets this means that two mere sets are identified if there is a bijection between them.

The model of multisets which we present in this paper is derived from a model of constructive set theory by Aczel 1978. The multiset model can in fact be seen as a description of what Aczel's model looks like through the eyes of type theory with the Univalence Axiom.

2 Notation and background

The following article is set with type theory as its intended metatheory. Some results depend on the Univalence Axiom, and are marked as such. Part of the article is formalised in Agda[6], in particular the more technical lemmas leading up to the extensionality theorem and the extensionality theorem itself. However, this article is self contained, and even the proofs for which there exists a formalisation are here presented in usual mathematical writing.

In ways of notation we will mostly use standard type theoretical notation, but written out as informal constructive mathematics in the style of the homotopy type theory book[7], rather than giving formal derivations. Our notation deviate a bit from the book in ways mentioned below. Mostly, these deviations take us close to how type theory is written in a formal proof system, such as Agda or Coq.

---

[5] Awodey, Pelayo, and Warren 2013, gives an exposition.

[6] Gylterud 2016, in the references contains a URL to the source code of the formalisation.

[7] Univalent Foundations Program 2013.

Application of functions and instantiation of dependent types are denoted by juxtaposition — e.g. $f\,a$ or $B\,a$ — leaving a small space between the function or type and their argument.

We use $A : \mathrm{Type}$ to denote that A is a type.

Definitions are signified by $:=$ with the type of the term often listed on the line above the definition. Definitional equalities are denoted by $\equiv$.

The equality sign $=$ is used to denote various equivalence relations — each time identified with a subscript, unless clear from the context. We will use the notation Id for the identity type. We denote disjoint union by $\vee$ or $+$, and binary products by $\wedge$ or $\times$.

We follow the book[8] in the definition of equivalence of types, $A \simeq B$, homotopy of functions, $f \sim g$, and notions such as *contractible*, *mere proposition*, *mere set*, and n-type. For the basic properties of these notions we refer the reader to (Univalent Foundations Program 2013).

Many proofs involve showing equivalences of types, which we strive to demonstrate, as far as possible, using chains of simpler type-algebra equivalences, such as $(\prod_{a:A} \sum_{b:B\,a} C\,a\,b) \simeq \left(\sum_{f:\prod_{a:A} B\,a} \prod_{a:A} C\,a\,(f\,a)\right)$.

It is is worth noting that we will consider quantifiers, such as $\forall$,$\exists$,$\prod$ and $\sum$ to bind weakly, so that for instance $\prod_{x:a} P\,x \to Q\,x$ disambiguates to $\prod_{x:a}(P\,x \to Q\,x)$ rather than $(\prod_{x:a} P\,x) \to Q\,x$. We sometimes will add the parenthesis to emphasise this.

3 The model

In this section we recall Aczel's model of constructive set theory, delve into homotopy type theory and construct a model of multiset theory.

3.1 Aczel's model

The idea behind the construction of Aczel's V type in Aczel 1978 is that, given a universe $U : \mathrm{Type}$ with decoding type $T : U \to \mathrm{Type}$, one can construct a setoid which captures the iteratively generated sets, where each set has an index of its elements in U. An element can be listed more than once in the index of the set. The equality relation of the setoid removes the distinction between equal sets with different representations, making sets equal if they have the same elements.

Definition A:1. Given a universe U, with decoding function $T : U \to \mathrm{Type}$, let Aczel's V be the setoid defined as follows.

[8] Univalent Foundations Program 2013.

V : Type
$V := W_{a:U} Ta$

$=_V : V \to V \to \text{Type}$
$$(\text{sup}\ \ a\ f) =_V (\text{sup}\ \ b\ g) := \left(\prod_{i:Ta} \sum_{j:Tb} (f\,i) =_V (g\,j)\right) \wedge \left(\prod_{j:Tb} \sum_{i:Ta} (f\,i) =_V (g\,j)\right)$$

The way to look at a canonical element $x \equiv (\text{sup}\ \ a\ f) : V$ is that a is a code for the index of elements in x and $f : Ta \to V$ picks out the elements (i.e. sets) contained in x.

Remark. A:2. Notice that the relation $=_V$ is U-small, if U has Σ-types and Π-types. That is, one can prove by W-induction that for each pair of elements $x, y : V$, there is a code in U for the type $x =_V y$. This is important, since we want to use equality to construct indices for new sets.

Definition A:3. Let elementhood in Aczel's V be defined as follows.

$$\in : V \to V \to \text{Type} \tag{1}$$
$$x \in (\text{sup}\ b\,g) = \sum_{i:Tb} x =_V (g\,i) \tag{2}$$

Remark. A:4. Since the relation $=_V$ is U-small, it follows that the relation $\in$ is also U-small.

Aczel goes on to prove that the setoid $(V, =_V)$, with the relation $\in$, is a model of Constructive Zermelo-Fraenkel set theory (CZF). In this paper, we take a different path, and ask the question: What is the nature of elements in V, without taking the quotient of $=_V$, and instead considering the identity type on V?

As noted, a set in V may have the same element listed several times, but the equality $=_V$ erases the distinction between representations that just differ by the number of times they repeat elements. However we cannot expect the identity type to do the same. Thus, we expect that the result will be more like multisets, possibly with the obstacle that order of elements may play a role. As we will see, this obstacle is overcome by the *univalence axiom*.

3.2 The identity type on W-types

The following result is due to Nils Anders Danielsson[9]. The result characterises the W-type of a type family $B : A \to \text{Type}$, in terms of the identity type of A and B, up to equivalence. The lemma does not make use of the Univalence axiom and can be carried out in plain Martin-Löf type theory.

A technical detail is that the proof as it stands, relies on η-reduction. The justification for this is that we take the function type in the W-type to be the Π-type of the logical framework, in which η-reduction is customary[10]. This is also how it is implemented in Agda. However one can carry out the proof without appeal to the η-reduction, as η-reduction holds up to provable equality (See page 62 of Nordström, Petersson, and Smith 1990).

Definition A:5. Given $A : \text{Type}$ and $B : A \to \text{Type}$, and an element $x : W_A B$ we denote by $\bar{x} : A$, and $\tilde{x} : B\bar{x} \to W_A B$ the operations given by $\overline{(\text{sup}\ a\ f)} \equiv a$ and $(\text{sup}\ a\ f) \equiv f$.

Lemma A:6. *For any* $A : \text{Type}$ *and* $B : A \to \text{Type}$*, and all* $x, y : W_A B$*, there is an equivalence*

$$Id_{W_A B}\, x\, y \simeq \sum_{\alpha : Id_A\, \bar{x}\, \bar{y}} Id\, \tilde{x}\, (B\alpha \cdot \tilde{y})$$

3.3 A model of multisets

We will now present our model of transfinite, iterative multisets, given a univalent universe $U : \text{Type}$ with decoding function $T : U \to \text{Type}$. It consists of a type M of multisets, an equality relation $=_M$ and a relation $\in$, which expresses elementhood. The type M is the same W-type as Aczel's V. The equality, however, is logically stricter than Aczel's equality, and, as we will show, equivalent in a strong sense to the identity type of M.

Definition A:7. We define

$$M : \text{Type by}$$
$$M := W_{a:U} Ta$$

[9] Danielsson 2012, only available on-line.
[10] Martin-Löf 1984.

and

$$=_M : M \to M \to \text{Type by}$$
$$(\sup\, a\, f) =_M (\sup\, b\, g) := \sum_{\alpha : Ta \simeq Tb} \prod_{x:Ta} (f x) =_M (g\,(\alpha\, x))$$

and

$$\in : M \to M \to \text{Type by}$$
$$x \in (\sup b\, g) := \sum_{i:Tb} x =_M (g\, i),$$

Remark: A:8. Observe that if U has Π-types, Σ-types and identity types, then $=_M$ and $\in$ are U-small, just like their corresponding relations in Aczel's V.

3.4 Equality and the identity type

Theorem A:9. *(UA) For each $x, y : M$, we have $(x =_M y) \simeq (Id_M\, x\, y)$.*

Proof. By W-induction. Assume $a, b : U$ and $f : Ta \to M$ and $g : Tb \to M$. Then

$$\begin{aligned}
(\sup a\, f) =_M (\sup b\, g) &\equiv \sum_{\alpha : Ta \simeq Tb} \prod_{x:Ta} (f x) =_M (g(\alpha x)) \\
\text{Induction hypothesis} \quad &\simeq \sum_{\alpha : Ta \simeq Tb} \prod_{x:Ta} \mathrm{Id}\,(f\, x)\,(g(\alpha x)) \\
\text{Definition of } \sim \quad &\equiv \sum_{\alpha : Ta \simeq Tb} f \sim g \cdot \alpha \\
\text{Extensionality} \quad &\simeq \sum_{\alpha : Ta \simeq Tb} \mathrm{Id}\, f\, (g \cdot \alpha) \\
\text{Univalence} \quad &\simeq \sum_{\alpha : a = b} \mathrm{Id}\, f\, (g \cdot T\alpha) \\
\text{Lemma A:6} \quad &\simeq \mathrm{Id}\,(\sup a\, f)\,(\sup b\, g)
\end{aligned}$$

□

The following lemma is important with respect to constructing multisets from logical formulas, in analogy to the comprehension axiom of set theory. We assume that our universe has $+$, Σ and Π-types, so if we can only prove that the base relations $=_M$ and $\in$ also live in the universe, then we can interpret all bounded first order formulas as families of types in U, indexed by some product of M with it self. Thus we have the following lemma:

Lemma A:10. *Id_M is essentially U-small, in the sense that for every $x, y : M$ there is an code $\iota\, x\, y : U$ such that $T(\iota\, x\, y) \simeq Id_M\, x\, y$.*

3.5 Extensionality

In set theory, the axiom of extensionality expresses that two sets are considered equal if they have the same elements. More precisely, two sets x and y are equal if for any z we have that $z \in x$ iff $z \in y$. This formulation of extensionality fails for multisets, but a very similar extensionality axiom may be formulated.

The principle of extensionality for multisets: *Two multisets x and y are considered equal if for any z, the number of occurrences of z in x and the number of occurrences of z in y are in bijective correspondence (in our symbolism: $z \in x \simeq z \in y$).*

We will now prove a strong version of this principle for our model. The crucial part of this is summarised in the following lemmas, concerning the fibres of functions.

Definition A:11. Given a function $f : A \to B$ we define Fibre $f : B \to$ Type by Fibre $f\, b := \sum_{a:A} \mathrm{Id}\,(f\, a)\, b$

Lemma A:12. *Given function extensionality, for any A, B, C : Type, and functions $f : A \to C$ and $g : B \to C$, the following equivalence holds:*

$$\left(\sum_{\alpha:A\to B} g \circ \alpha \sim f\right) \simeq \left(\prod_{c:C} \text{Fibre } f\, c \to \text{Fibre } g\, c\right) \tag{3}$$

Proof. We define the maps γ and δ as follows:

$$\gamma : \left(\sum_{\alpha:A\to B} g \circ \alpha \sim f\right) \to \left(\prod_{c:C} \text{Fibre } f\, c \to \text{Fibre } g\, c\right)$$
$$\gamma\,(\alpha, \sigma)\, c\,(a, p) := (\sigma_a, \sigma_a \cdot p)$$

$$\delta : \left(\prod_{c:C} \text{Fibre } f\, c \to \text{Fibre } g\, c\right) \to \left(\sum_{\alpha:A\to B} g \circ \alpha \sim f\right)$$
$$\delta\, F := (\lambda a.\pi_0(F\,(f\, a)(a, \mathrm{refl}_a)), \lambda a.\pi_1(F\,(f\, a)\,(a, \mathrm{refl}_a)))$$

Unfolding the definitions shows that $\delta\,(\gamma\,(\alpha, \sigma)) \equiv (\alpha, \sigma)$ (up to η-reduction). Id-induction on the fibres of f shows that $\gamma\,(\delta\, F) \sim F$. Thus, by function extensionality, we have the desired equivalence. □

Lemma A:13. *Given function extensionality, for any A, B, C : Type, and functions $f : A \to C$ and $g : B \to C$, the following equivalence holds:*

$$\left(\sum_{\alpha : A \simeq B} g \circ \alpha \sim f\right) \simeq \left(\prod_{c:C} \text{Fibre } f\, c \simeq \text{Fibre } g\, c\right) \tag{4}$$

Proof. The proof goes by showing that the equivalence constructed in A:12 preserves equivalences. Since being an equivalence is a (-1)-type, the resulting restriction of A:12 to equivalences is again an equivalence.

Let γ and δ be as in A:12, and denote by γ' and δ' the same construction, but with f and g having exchanged roles. First step is to show that for all $\alpha : A \to B$ and $\sigma : g \circ \alpha \sim f$, if α is an equivalence, then for every $c : C$ the function $\gamma\,(\alpha, \sigma)\, c$: Fibre $f\, c \to$ Fibre $g\, c$ is an equivalence. Let $F_c := \gamma\,(\alpha, \sigma)\, c$. We construct the inverse $F_c^{-1} := \gamma'\,(\alpha^{-1}, \sigma')$, where $\sigma' : f \circ \alpha^{-1} \sim g$ is the proof obtained by reversing $\sigma\alpha^{-1} : g \circ \alpha \circ \alpha^{-1} \sim f \circ \alpha^{-1}$ and composing with the proof that $g \circ \alpha \circ \alpha^{-1} \sim g$. That F_c^{-1} is indeed an inverse of F_c can be verified by Id-induction on the fibres of f and g respectively.

We then show that for all F such that $F\, c$: Fibre $f\, c \to$ Fibre $g\, c$ is an equivalence for all $c : C$, the function $\pi_0(\delta\, F) : A \to B$ is an equivalence. Let $\alpha := \pi_0(\delta\, F)$. Its inverse is given by $\alpha^{-1} := \pi_0(\delta'(F^{-1}))$, and the fact that it is an inverse of α stems from the fact that for any $a : A$ and $c : C$ and $h : \text{Id}\,(f\, a)\, c$ we have that the transport of $(a, \text{refl}_{f\,a})$: Fibre $f\,(f\, a)$ along h is (a, h), and likewise for g.

□

Theorem A:14. *(UA) Given $x, y : M$, the following equivalence holds.*

$$(x =_M y) \simeq \prod_{z:M} (z \in x \simeq z \in y) \tag{5}$$

Proof. From Theorem A:9 we deduce that $x \in (\sup\, A\, f) \simeq$ Fibre $x\, f$. This allows us to reformulate the above equivalence to be an instance of Lemma A:13.

□

4 Multiset constructions

Aczel's V is a model of CZF. To mirror this we look at axioms of constructive set theory, and attempt to find corresponding axioms for mul-

tisets. The main observation is that definite axioms[11] can be systematically changed to axioms which makes sense for multisets, by carefully strengthening logical equivalence, $\leftrightarrow$, to equivalence in type theory $\simeq$. A feature of this conversion is that for the axioms below, we can retain the constructions from Aczel's V when we prove that the changed axioms hold for M.

The axioms we will have a look at are

- Extensionality
- Restricted separation
- Union Replacement
- Pairing & singletons
- Infinity.
- Exponentiation / Fullness
- Collection.

4.1 Restricted separation

The axiom of restricted separation says that we can select subsets by use of formulas, as long as they are bounded. For multisets, this corresponds to that we may multiply the number of occurrences by the family of sets represented by the formula, as long as the family is U-small.

$$(\texttt{RSEP}) \qquad \forall x \exists u \forall z \, (z \in u \leftrightarrow (z \in x \wedge P))$$

where P is a restricted formula where u does not occur freely in P.

The formulation of RSEP in first order logic can be translated into type theory, given our domain M, replacing $\leftrightarrow$ with $\simeq$.

Proposition A:15.

$$(\texttt{M} - \texttt{RSEP}) \qquad \prod_{x:M} \sum_{u:M} \prod_{z:M} (z \in u \simeq (z \in x \wedge T\,(P\,z)))\,,$$

where $P : M \to U$, is a U-small family.

[11] An axiom is *definite* if any set it claims existence of is characterised uniquely.

Proof. Define

$$\text{Sep} : (M \to U) \to M \to M \tag{6}$$

$$\text{Sep}\ P\, x := \sup \left(\sum_{i:T\,\bar{x}} P\,(\tilde{x}\,i) \right)\ (\tilde{x} \circ \pi_0) \tag{7}$$

We must show that for all x and P, that for every z we have $z \in \text{Sep}\ P\, x \simeq z \in x \wedge T\,(P\, z)$.

We have:

$$\begin{aligned}
z \in \text{Sep}\ P\, x &\equiv \sum_{p:\sum_{i:T\,\bar{x}} T\,(P\,(\tilde{x}\,i))} \tilde{x}(\pi_0\, p) =_M z && (8)\\
&\simeq \sum_{i:T\,\bar{x}} \sum_{q:T\,(P\,(\tilde{x}\,i))} \tilde{x}(\pi_0\,(i,q)) =_M z && (9)\\
&\equiv \sum_{i:T\,\bar{x}} \sum_{q:T\,(P\,(\tilde{x}\,i))} \tilde{x}\,i =_M z && (10)\\
&\equiv \sum_{i:T\,\bar{x}} (T\,(P\,(\tilde{x}\,i)) \wedge \tilde{x}\,i =_M z) && (11)\\
&\simeq \sum_{i:T\,\bar{x}} (\tilde{x}\,i =_M z \wedge T\,(P\,(\tilde{x}\,i))) && (12)\\
&\simeq \sum_{i:T\,\bar{x}} (\tilde{x}\,i =_M z \wedge T\,(P\, z)) && (13)\\
&\simeq \left(\sum_{i:T\,\bar{x}} \tilde{x}\,i =_M z \right) \wedge T\,(P\, z) && (14)\\
&\equiv z \in x \wedge T\,(P\, z) && (15)
\end{aligned}$$

□

4.2 Union-replacement

In Aczel and Rathjen 2001, the authors introduce the axiom of Union-Replacement. We use this axiom instead of separate union and replacement axioms as it seems a more natural construction to use. For multisets it says that if we have a family of multisets, indexed by a multiset, we can take their multiset union.

$$\begin{aligned}
(\texttt{UR}) \qquad & \forall a\ (\forall x \in a\ \exists b\, \forall y\ (y \in b \leftrightarrow Q(x,y)) \\
& \to \exists c\, \forall y\ (y \in c \leftrightarrow \exists x \in a\, Q(x,y)))
\end{aligned}$$

Rendering this in type theory and applying the translation to multisets we get:

Proposition A:16.

$$(\mathtt{M}-\mathtt{UR}) \qquad \prod_{a:M}\left(\prod_{i:T\bar{a}}\sum_{b:M}\prod_{y:M}(y\in b\simeq Q\,(\tilde{a}\,i)\,y)\right) \to \sum_{c:M}\prod_{y:M}\left(y\in c\simeq\sum_{i:T\bar{a}}Q(\tilde{a}\,i)\,y\right)$$

where $Q : V \to V \to Set$ is any relation.

Proof. We define

$$\text{UnionRep} : (a : M) \to (\bar{a} \to M) \to M \tag{16}$$

$$\text{UnionRep}\ a\ f := \sup\left(\sum_{i:T\bar{a}}\overline{(f\,i)}\right)\left(\lambda p.\widetilde{f\,(\pi_0\,p)}\,(\pi_1\,p)\right) \tag{17}$$

Let us fix $a : M$. Then, from the assumption of $\prod_{i:T\bar{a}}\sum_{b:M}\prod_{y:M}(y\in b\simeq Q\,(\tilde{a}\,i)\,y)$, we can extract $f : T\bar{a} \to M$ such that for all $i : T\bar{a}$ and all $y : M$ we get $y\in(f\,i)\simeq Q\,(\tilde{a}\,i)\,y$. What we then need, is to show that for every $y : M$ we have $y\in(\text{UnionRep}\ a\,f)\simeq\sum_{i\in T\bar{a}}Q(\tilde{a}\,i)\,y$.

We have

$$y\in(\text{UnionRep}\ a\,f)\equiv\sum_{p:\sum_{i:T\bar{a}}\overline{f\,i}}\widetilde{f\,(\pi_0\,p)}\,(\pi_1\,p)=_M y \tag{18}$$

$$\simeq\sum_{i:T\bar{a}}\sum_{j:T\overline{f\,i}}\widetilde{(f\,i)}\,j=_M y \tag{19}$$

$$\equiv\sum_{i:T\bar{a}}y\in(f\,i) \tag{20}$$

$$\simeq\sum_{i:T\bar{a}}Q(\tilde{a}\,i)\,y \tag{21}$$

□

4.3 Singletons

In set theory, singletons are usually constructed by pairing an element with itself. For multisets defining singletons from pairs would not work, as the resulting multiset would contain the element twice, not once. We therefore will prove that our model has singletons.

If we were to have a singleton axiom in set theory, it would look like:

$$(\texttt{SING}) \qquad \forall a\, \exists b\, \forall z\ (z \in b \leftrightarrow z = a)$$

Which for our multisets becomes:

Proposition A:17.

$$(\texttt{M} - \texttt{SING}) \qquad \prod_{a:M} \sum_{b:M} \prod_{z:M} (z \in b \simeq z =_M a)$$

Proof. We define

$$\begin{aligned} &\text{Sing} : M \to M \\ &\text{Sing}\ a := \sup\ 1\ (\lambda i.a) \end{aligned}$$

and prove that for every $a, z : M$ we have $(z \in \text{Sing}\ a) \simeq (z =_M a)$.

$$\begin{aligned} z \in \text{Sing}\ a &\equiv \sum_{i:1} (\lambda i.a)\ i =_M z && (22)\\ &\equiv \sum_{i:1} a =_M z && (23)\\ &\equiv 1 \wedge (a =_M z) && (24)\\ &\simeq z =_M a && (25) \end{aligned}$$

□

Notation: We will from now on use the notation $\{a\} := \text{Sing}\ a$.

4.4 Pairing

The axiom of pairing,

$$(\texttt{PAIR}) \qquad \forall a\, \forall b\, \exists c\, \forall z\ (z \in b \leftrightarrow (z = a \vee z = b))\,,$$

becomes

Proposition A:18.

$$(\texttt{M} - \texttt{PAIR}) \qquad \prod_{a:M} \prod_{b:M} \sum_{c:M} \prod_{z:M} (z \in c \simeq (z =_M a \vee z =_M b))$$

Proof. First, define

$$p : M \to M \to 2 \to M \tag{26}$$
$$p\, a\, b\, (l\, *) := a \tag{27}$$
$$p\, a\, b\, (r\, *) := b \tag{28}$$

Then, let us define

$$\text{Pair} : M \to M \to M$$
$$\text{Pair}\ a\, b := \sup\ 2\ (p\, a\, b)$$

It remains to show that for all $a, b, z \in M$ we have $z \in (\text{Pair}\ a\, b) \simeq (z =_M a \vee z =_M b)$.

$$z \in (\text{Pair}\ a\, b) \equiv \sum_{i:2} p\, i =_M z \tag{29}$$
$$\simeq p\, (l\, *) =_M z \vee p\, (r\, *) =_M z \tag{30}$$
$$\equiv a =_M z \vee b =_M z \tag{31}$$
$$\simeq z =_M a \vee z =_M b \tag{32}$$

□

Notation: We will from now on use the notation $\{a, b\} := \text{Pair}\ a\, b$.

Lemma A:19. *For all $a, b, a', b' : M$,*

$$\{a\} =_M \{a'\} \simeq a =_M a' \tag{33}$$
$$\{a, b\} =_M \{a', b'\} \simeq \left((a = a' \wedge b = b') \vee (a = b' \wedge b = a')\right). \tag{34}$$

Example A:20. The singleton with two elements

Observe that $(\{\emptyset, \emptyset\} = \{\emptyset, \emptyset\}) \simeq 2$. This leads to the perhaps surprising fact that

$$(\{\emptyset, \emptyset\} \in \{\{\emptyset, \emptyset\}\}) \simeq 2. \tag{35}$$

This might leave us feeling a bit uneasy, as this is supposed to be a singleton, not a "doubleton", but we will see later (Example A:29) how to construct a multiset in which $\{\emptyset, \emptyset\}$ occurs but once, and that this construction is somewhat like a quotient of the singleton. For the time being we accept this slight anomaly as a consequence of our rules.

Remark: A:21. Using singletons, pairs and union we can construct any finite tupling of elements of M. As we see from the binary case (A:19), the induced mapping $M^n \to M$ is not an embedding.

4.5 Ordered Pairs

In set theory there are many equivalent ways to encode ordered pairs from unordered pairs. The most common one, the Kuratowski encoding, defines $\langle a, b\rangle = \{\{a\}, \{a, b\}\}$. It satisfies the characteristic property, that for all a, b, a', b',

$$\langle a, b\rangle = \langle a', b'\rangle \leftrightarrow (a = a' \wedge b = b') \tag{36}$$

To understand ordered pairs of multisets, we should therefore require that for all $a, b, a', b' : M$

$$\langle a, b\rangle = \langle a', b'\rangle \simeq (a =_M a' \wedge b =_M b') \tag{37}$$

However, this is not satisfied by mimicking the Kuratowski encoding. We can see this by letting $a = b = a' = b = \emptyset$. Given this, we can calculate that $\{\{a\}, \{a, b\}\} =_M \{\{a'\}, \{a', b'\}\} \simeq 2$, while
$(a =_M a' \wedge b =_M b') \simeq 1$.

In stead of the Kuratowski encoding, we use the older definition of Wiener 1914, $\langle a, b\rangle = \{\{\{a\}, \emptyset\}, \{\{b\}\}\}$, which harmonises with the multiset version of the characteristic property.

Proof. Observe that $\{\{a\}, \emptyset\} \neq_M \{\{b\}\}$ and $\{a\} \neq_M \emptyset$. So we get that

$$\begin{aligned}
\langle a, b\rangle =_M \langle a', b'\rangle &\equiv \{\{\{a\}, \emptyset\}, \{\{b\}\}\} =_M \{\{\{a'\}, \emptyset\}, \{\{b'\}\}\} && (38)\\
&\simeq \{\{a\}, \emptyset\} =_M \{\{a'\}, \emptyset\} \wedge \{\{b\}\} =_M \{\{b'\}\} && (39)\\
&\simeq (\{a\} =_M \{a'\} \wedge \emptyset =_M \emptyset) \wedge (\{b\} =_M \{b'\}) && (40)\\
&\simeq a =_M a' \wedge b =_M b' && (41)
\end{aligned}$$

□

4.6 Cartesian products

We obtain the cartesian product of two multisets by nesting UnionRep around pairing.

Definition A:22. Given $a, b : M$ define

$$a \times b := \text{UnionRep}\ a\,(\lambda i.\,\text{UnionRep}\ b\,(\lambda j.\langle \tilde{a}\,i, \tilde{b}\,j\rangle)) \tag{42}$$

Each pairing can occur multiple times in the cartesian product. To be precise $\langle x, y\rangle \in (a \times b) \simeq (x \in a) \times (y \in b)$.

4.7 Functions

There are several choices one could make as to what constitutes a function between multisets. Just like in set theory, where a function between sets is it self a set — namely a set of pairs — we should like functions between multisets themselves to be multisets. Therefore, a given pair $\langle x, y\rangle$ cannot occur an unbounded number of times in a function, as we then would have problems collecting functions into exponential multisets.

The weakest notion of a function between two multisets is just a map of occurrences. We will refer to this as a "multiset operation". Often one consider the stricter notion which sends equal occurrences to equal occurrences. This notion we will denote by "multiset function". As we will now see, we can express both in our model.

In the model, the notion of a multiset operation between sup $A\,f$ and sup $B\,g$ corresponds exactly to a map $\phi : A \to B$, and the stricter notion adds the requirement that if $f\,a =_M f\,a'$ then $g\,(\phi\,a) =_M g\,(\phi\,a')$. The stricter notion corresponds to functions in Aczel's V, but the weaker notion is equivalent to the stricter notion in the case of sup $A\,f$ and sup $B\,g$ being sets, in the sense of A and B being of type level 0 and f and g being injections. Therefore, we can consider both an extension of the notion of function to multisets.

The question we will now entertain is: How to capture these two notions in the kind of formulas we have so far considered for other axioms? Starting with the operations, we remind our selves that, for iterative sets, a function $f : A \to B$ is a subset of $A \times B$, such that the projection down to A is a bijection. Having equivalences available in our language, we can use the fibrewise equivalence lemma[12] to describe the corresponding situation for multisets.

Definition A:23. We define the weak notion of a multiset function as

[12] Lemma A:13

follows.

$$\begin{aligned}
&\text{Operation} : M \to M \to M \to Set \\
&\text{Operation}_{a\,b}\ f := \left(\prod_z z \in f \to \sum_x \sum_y z = \langle x, y\rangle\right) \\
&\qquad\qquad \wedge \left(\prod_x x \in a \simeq \sum_y \langle x, y\rangle \in f\right) \\
&\qquad\qquad \wedge \left(\prod_y y \in b \leftarrow \sum_x \langle x, y\rangle \in f\right)
\end{aligned}$$

Observe that weakening the $\simeq$ to $\leftrightarrow$ does not give the usual definition of a function for sets, but rather that of a total binary relation. Total binary relations form a set in classical set theory, but in CZF this is weakened to the subset collection axiom which states that there is a set of total relations in which every total relation has a refinement. We will later prove that the collection of multiset operations form a multiset, and this should be seen a form of subset collection / fullness.

Definition A:24. We define the notion of a multiset function as follows.

$$\begin{aligned}
&\text{Function} : M \to M \to M \to Set \\
&\text{Function}_{a\,b}\ f := \text{Operation}_{a\,b}\ f \wedge \prod_{x,x',y,y'} (\langle x, y\rangle \in f \wedge \langle x', y'\rangle \in f) \\
&\qquad\qquad\qquad\qquad\qquad\qquad \to x = x' \to y = y'
\end{aligned}$$

4.8 Fullness, subset collection and operations

In constructive set theory the axiom of fullness states that for each pair of sets a,b there is a set of total relations from a to b such that any total relation has a restriction to these. The equivalent (relative to the rest of the axioms of CZF) axiom of subset collection is a variation of fullness which avoids the complication of using pairs to encode relations.

$$\begin{aligned}
(\texttt{SUB}-\texttt{COLL}) \quad &\forall a, b \exists u \forall v (\forall x \in a \exists y \in b Q(x, y) \\
&\to \exists z \in u (\forall x \in a \exists y \in z Q(x, y) \wedge \forall y \in z \exists x \in a Q(x, y)))
\end{aligned}$$

An unfortunate feature of the subset collection axiom is that it states the existence of certain sets without defining them uniquely. Classically, the sets of functions would satisfy the property of fullness. In fact, the requirement that the set of functions satisfying the fullness property is

equivalent to the axiom of choice. This raises the question of what the constructive nature of this set really is.

Some insight on the matter can be found by studying Aczel's model of CZF in type theory. There the underlying type of the subset collection set between sup $A\,f$ and sup $B\,g$ is the function type $A \to B$. In other words, the subset collection sets are sets of operations. However, the first order language of set theory is extensional, and thus unable to exactly pin down what an operation is, thus the sets of which the axiom claim existence are left indefinite by the axiom itself. In this respect, the axiom for multisets, in our language where we have borrowed the connective $\simeq$ from type theory, stating the existence of multisets of operations is a refinement of collection/fullness into a definite axiom, namely *exponentiation for operations*.

4.9 Exponentiation

We define the exponential of two multisets.

Definition A:25. Let $a, b : M$ be multisets and define

$$\begin{aligned}&\mathrm{Exp}\ a\,b : M\\ &\mathrm{Exp}\ a\,b := \sup\,(\bar{a} \to \bar{b})\,(\lambda f.\,\sup\,\bar{a}\,(\lambda i.\langle \tilde{a}i, \tilde{b}(fi)\rangle))\end{aligned}$$

Next, we formulate the exponentiation axiom for multisets and prove that there exists a multiset in our model satisfying this axiom.

Proposition A:26.

$$(\mathtt{M-EXP}) \qquad \prod_{a:M}\prod_{b:M}\sum_{c:M}\prod_{z:M}\,(z \in c \simeq (\mathrm{Operation}_{a\,b}\,z))$$

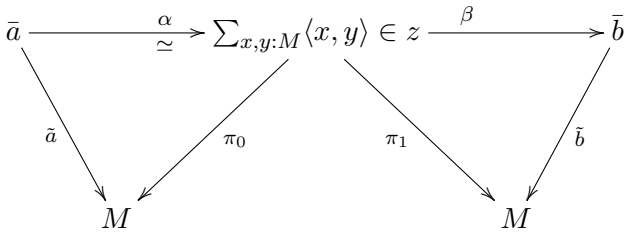
Proof. Let c be Exp $a\,b$. Thus, we need to prove that for any given $z : M$, there is an equivalence $\mathrm{Operation}_{a\,b}\ z \simeq z \in \mathrm{Exp}\ a\,b$. We will give this equivalence in two steps. (A heuristic reason for why we need to jump through a hoop here is that $\mathrm{Operation}_{a\,b}\ z$ has three factors while $z \in \mathrm{Exp}\ a\,b$ has two (dependent ones), and we cannot construct the equivalence factorwise. We therefore construct a more finely grained equivalent which maps factorwise to both.)

Step 1. The type $\mathrm{Operation}_{a\,b}\ z$ is equivalent to the following data:

- $\alpha : \bar{a} \to \sum_{x,y:M}\langle x, y\rangle \in z$
- $\beta : \sum_{x,y:M}\langle x, y\rangle \in z \to \bar{b}$

- $\epsilon : \alpha \circ \pi_0 = \tilde{a}$, where $\pi_0 : \sum_{x,y:M} \langle x, y\rangle \in z \to M$ extracts the x component.
- $\delta : \beta \circ \tilde{b} = \pi_1$, where $\pi_1 : \sum_{x,y:M} \langle x, y\rangle \in z \to M$ extracts the y component.
- $\pi_0 \circ \pi_1 \circ \pi_1 : \sum_{x,y:M} \langle x, y\rangle \in z \to \bar{z}$, which extracts the z-index, is an equivalence.

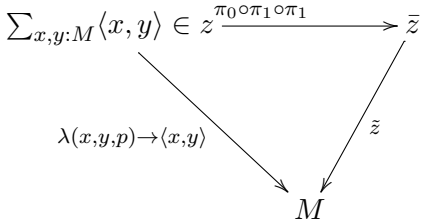
This data can be succinctly expressed by the following commutative diagram:



(43)

The type $\text{Operation}_{a\,b}\ z$ is a product of three factors. The first factor is equivalent to $\pi_0 \circ \pi_1 \circ \pi_1 : \sum_{x,y:M} \langle x, y\rangle \in z \to \bar{z}$ being an equivalence, since it says that all elements of z are pairs. The second factor is equivalent to the data α and ϵ above, by the fibrewise equivalence lemma (Lemma A:13). Similarly, the third factor is equivalent to the data β and δ. Thus, we conclude that $\text{Operation}_{a\,b}\ z$ is indeed equivalent to the above data.

Step 2. The data given in step 1 is equivalent to $z \in \text{Exp}\ a\, b$, which is to say that z is equal to the graph of a map $\bar{a} \to \bar{b}$.



(44)

Given the data in step 1, define $f : \bar{a} \to \bar{b}$ by $f := \beta \circ \alpha$. Since $\pi_0 \circ \pi_1 \circ \pi_1 : \sum_{x,y:M} \langle x, y\rangle \in z \to \bar{z}$ is an equivalence, which makes the diagram (44) commute, we derive that

$$z = \sup\, \bar{a}\,(\lambda i.\langle \pi_0\,(\alpha\, i), \pi_0\,(\pi_1\,(\alpha\, i))\rangle) \tag{45}$$
$$= \sup\, \bar{a}\,(\lambda i.\langle \tilde{a}\, i, \tilde{b}\,(\beta\,(\alpha\, i))\rangle) \tag{46}$$
$$= \sup\, \bar{a}\,(\lambda i.\langle \tilde{a}\, i, \tilde{b}\,(f\, i)\rangle) \tag{47}$$

which is precisely that z is the graph of f.

In the other direction, assuming that $z = \sup\, \bar{a}(\lambda i.\langle \tilde{a}\, i, \tilde{b}\,(f\, i)\rangle$, we observe that $\pi_0 \circ \pi_1 \circ \pi_1 : \sum_{x,y:M} \langle x, y\rangle \in z \to \bar{z}$ is in fact an equivalence, and project the equivalence $q : \bar{z} \simeq \bar{a}$ from the assumed equality. We then factor f into $\alpha := (q \circ \pi_0 \circ \pi_1 \circ \pi_1)^{-1}$ and $\beta := f \circ q \circ \pi_0 \circ \pi_1 \circ \pi_1$. The fact that the rest of the equalities of the data hold, follows from the definition of α and β and that diagram (44) commutes.

That this construction is an equivalence is (tedious) routine verification, from which we spare the reader.

In conclusion, combining the above two steps, we have constructed an equivalence $\mathrm{Operation}_{a\,b}\ z \simeq z \in \mathrm{Exp}\ a\, b$.

□

4.10 Natural numbers

The natural number axiom is straightforward to translate, and the construction is exactly the same as in Aczel's model.

Applying the usual abbreviations,

$$S\, y\, z \equiv \forall x\,(x \in z \leftrightarrow (x \in y \vee x = y))$$

which codes the relation z is the successor of y, and

$$Z\, z \equiv \forall x\ \neg x \in z,$$

coding z is zero – the axiom of infinity in set theory can be expressed as:

$$(\texttt{INF}) \qquad \exists u \forall z\,(z \in u \leftrightarrow (Z\, z \vee \exists y \in u\ S\, y\, z))$$

For multisets we give similar definitions of S and Z, in order to define an axiom of infinitity.

$$S\, y\, z := \prod_{x:M} (x \in z \cong (x \in y \vee x = y))$$
$$Z\, z := \prod_{x:M} \neg x \in z$$

Proposition A:27.

$$(\mathtt{M}-\mathtt{INF}) \qquad \sum_{u:M}\prod_{z:M}\left(z\in u\cong\left(Z\,z\vee\sum_{y:M}y\in u\wedge S\,y\,z\right)\right)$$

Proof. The construction of a natural number object for M is the same as the construction for Aczel's V. We define a sequence of multisets $N:\mathbb{N}\to M$, by

$$\begin{aligned}N\,0&:=\emptyset\\ N\,(n+1)&:=N\,n\cup\{N\,n\}\end{aligned}$$

And let our natural number object be $u=\sup\mathbb{N}\,N$. It just remains to observe that u satisfies the condition of $\mathtt{M}-\mathtt{INF}$

$$\begin{aligned}z\in u&\equiv\sum_{n:\mathbb{N}}N\,n=z\\ &\cong z=\emptyset\vee\sum_{n:\mathbb{N}}N\,(Sn)=z\\ &\cong z=\emptyset\vee\sum_{n:\mathbb{N}}(N\,n\cup\{N\,n\})=z\\ &\cong Z\,z\vee\sum_{n:\mathbb{N}}S\,(N\,n)\,z\\ &\cong Z\,z\vee\sum_{y:M}y\in m\wedge S\,y\,z\end{aligned}$$

□

5 Homotopic aspects of M

The previous section might seem as though not much have changed going from the sets V to the multisets M. In this subsection we will take a look at what objects might be in M for which, since we work with the identity type on M, higher homotopies come into play. First of all, we observe that M has the same number of non-trivial levels of homotopy as U has.

5.1 Homotopy n-type

Recall from the book "Homotopy Type Theory"[13] that types can be divided into levels, according to how many times one can iterate the

[13]Univalent Foundations Program 2013.

identity type on the type before it becomes trivial, in the sense of being contractible. A type is contractible if it has an element, which every other element is (uniformly) equal to. This is captured by the following definitions.

$$\text{isContractible}(X) := \sum_{x:X} \prod_{x':X} x' = x \tag{48}$$

$$\texttt{is}{-}(-2){-}\texttt{type}\ X = \text{isContractible}(X) \tag{49}$$

$$\texttt{is}{-}(n+1){-}\texttt{type}\ X := \prod_{x,y:X} \texttt{is}{-}(n){-}\texttt{type}(\mathrm{Id}\, x\, y) \tag{50}$$

Proposition A:28. *M has the same homotopy n-type as U.*

Proof. If M is homotopy n-type, then U is also homotopy n-type. This follows from the fact that the following map is an embedding.

$$\iota : U \to M \tag{51}$$

$$\iota a := (\sup\ a\ (\lambda a.\emptyset)) \tag{52}$$

On the other hand if U has homotopy n-type, then we show by W-induction on M that M also has homotopy n-type.

Let $x = (\sup\ a\ f)$ and $y = (\sup\ b\ g)$, and consider $\mathrm{Id}\, x\, y$. By Lemma A:6 we know that:

$$Id\, x\, y \simeq \sum_{\alpha:\mathrm{Id}_U\, a\, b} \mathrm{Id}_{Ta \to W} f\ (B\alpha \cdot g) \tag{53}$$

From W-induction we have the induction hypothesis that the image of f has homotopy n-type, and by Theorem 7.1.8 in the book[14]., we know that this Σ-type also has homotopy n-type,

□

5.2 HITs and multisets

If our universe has Higher Inductive Types (HITs), we can construct multisets where the index set is a higher groupoid structure. An interesting fact is that even if $\bar{a}$ is a higher groupoid, we can still have that $x \in a$ is 1-type for all x .

[14]Univalent Foundations Program 2013.

Example A:29. In Example A:20 we saw that the singleton construction unexpectedly gave singletons where the single element occurred twice, because of its non-trivial equalities to itself in M. A solution to this is to take the connected compontent of the element in M as the index set of the singleton, instead of just 1, along with the inclusion into M. However, this requires the connected component to be U-small, which the usual construction does not guarantee. Adding that assumption, which we conjecture could hold in general (in homotopical models), since M is locally U-small, we can construct singletons even for elements of M with non-trivial self-identities.

For any multiset $x : M$, such that there is $t : U$ which represents the connected component of x, i.e. $\alpha : T\,t \simeq \sum_{y:M} \|x =_M y\|_{-1}$, we can define a the singleton $s\,x\,t\,\alpha := \sup\, t(\pi_0 \circ \alpha)$. The map $\pi_0 \circ \alpha$ is an embedding, since $\|x =_M y\|_{-1}$ is a mere proposition. It follows that the fibres $y \in s\,x\,t\,\alpha$ are all propositions, and in particular $x \in s\,x\,t\,\alpha$ is contractible.

References

Aczel, Peter (1978). "The Type Theoretic Interpretation of Constructive Set Theory". In: *Logic Colloquium '77*. Ed. by A. MacIntyre, L. Pacholski, and J. Paris. North–Holland, Amsterdam-New York, pp. 55–66.

Aczel, Peter and Michael Rathjen (2001). *Notes on Constructive Set Theory*. Tech. rep. Institut Mittag-Leffler.

Awodey, S., Á. Pelayo, and M. A. Warren (2013). "Voevodsky's Univalence Axiom in homotopy type theory". In: *ArXiv e-prints*. arXiv: `1302.4731 [math.HO]`.

Blizard, Wayne D. (1988). "Multiset theory." In: *Notre Dame Journal of Formal Logic* 30.1, pp. 36–66. DOI: `10.1305/ndjfl/1093634995`. URL: `http://dx.doi.org/10.1305/ndjfl/1093634995`.

Danielsson, Nils Anders (2012). *Positive h-levels are closed under W*. URL: `https://homotopytypetheory.org/2012/09/21/positive-h-levels-are-closed-under-w/`.

Gylterud, Håkon Robbestad (2016). *Formalisation of iterative multisets and sets in Agda*. URL: `http://staff.math.su.se/gylterud/agda/`.

Martin-Löf, Per (1984). *Intuitionistic type theory. Notes by Giovanni Sambin*. Vol. 1. Studies in Proof Theory. Bibliopolis, Naples, pp. iv+91. ISBN: 88-7088-105-9.

Nordström, B., K. Petersson, and J. M. Smith (1990). *Programming in Martin-Löf's Type Theory*.

Rado, Richard (1975). "The cardinal module and some theorems on families of sets". In: *Annali di Matematica Pura ed Applicata* 102.1, pp. 135–154. ISSN: 1618-1891. DOI: 10.1007/BF02410602. URL: http://dx.doi.org/10.1007/BF02410602.

Univalent Foundations Program, The (2013). *Homotopy Type Theory: Univalent Foundations of Mathematics*. Institute for Advanced Study: homotopytypetheory.org. URL: http://homotopytypetheory.org/book.

Wiener, Norbert (1914). "A Simplification of the Logic of Relations". In: *Proceedings of Cambridge Philosophical Society* 17, pp. 387–390.

B

From multisets to sets in Homotopy Type Theory

Abstract

We give a model of set theory based on multisets in homotopy type theory. The equality of the model is the identity type. The underlying type of iterative sets can be formulated in Martin-Löf type theory, without Higher Inductive Types (HITs), and is a sub-type of the underlying type of Aczel's 1978 model of set theory in type theory. The Voevodsky Univalence Axiom and mere set quotients (a mild kind of HITs) are used to prove the axioms of constructive set theory for the model. We give an equivalence to the model provided in Chapter 10 of "Homotopy Type Theory" by the Univalent Foundations Program.

1 Introduction

The first model of set theory in type theory is due to Aczel and models the constructive set theory CZF[15]. The underlying type of sets in this model is $W_U T$, the type of all well-founded trees with branchings in a universe U with decoding family $T : U \to \text{Type}$. The interpretation of equality in this model allows deduplication and permutation of subtrees — incorporating the intuition that the order and multiplicity of elements of a set are irrelevant. If we instead insist to interpret equality as the identity type and assume the univalence axiom, the underlying type no longer models set theory, but rather multiset theory.

In a related work[16] we explore the notion of iterative multisets in type theory. Here we will specialise from the general multisets to the set-like ones, where each element occurs at most once. We will start by summarising the model of multisets and its relation to Aczel's model of CZF in type theory[17].

Once the notion of a multiset is defined, it is natural to study the hereditary subtype of multisets where each element occurs at most once. These are in a certain sense the most natural representations of iterative sets from a homotopy type theory point of view. These are namely the

[15] Aczel 1978.
[16] Part A
[17] Aczel 1978.

multisets for which the element hood relation is hereditarily, merely propositional (type level -1).

In this text we explore how this type models various axioms of constructive set theory. We also show that it is equivalent to the higher inductive type outlined in the book "Homotopy Type Theory"[18].

1.1 From multisets to sets – two ways

Assume for a moment that we would like to explain the notion of multiset to someone who knows the notion of a set. Here are two similar attempts at such an explanation.

1.1.1 Variant A

> A multiset is a generalisation of sets in which an element may occur any number of times, not just at most once.

1.1.2 Variant B

> A multiset is like a set with the extra information attached to each element about how many times it occurs in the multiset.

The two descriptions are almost identical, but critically different. Variant A describes the multisets as a more general concept than sets, in the sense that a set is a *special case* of a multiset, namely those multisets in which each element occurs at most once. Variant B, on the other hand, describes a multisets as sets with some extra structure. In both cases the impression given is that the multisets are, in an informal way, a *bigger concept* than that of a set. Variant A says that sets are just some of the multisets, while Variant B says that for each set there are numerous ways to make a multiset from it.

In mathematics these two notions of "bigger" correspond to the notion of

- sets being a subtype of multisets (Variant A)
- sets being a quotient of multisets (Variant B)

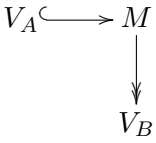
If we were to turn the direction of explanations, making them explanations of the concept of a set from the notion of a multiset, the above suggests two distinct routes of constructing a notion of sets from a notion of multisets. We either identify the subtype of *set-like* multisets,

[18] Univalent Foundations Program 2013.

or identify the equivalence relation of multisets which *forgets the extra structure of multiplicites.*

Example B:1. The multiset $\{a, a, b\}$ would not be considered a set-like in the spirit of Variant A, since a occurs twice. On the other hand, Variant B would contest that $\{a, a, b\}$ and $\{a, b, b\}$ would *represent the same* set, since in both cases a and b are exactly the elements occurring at least once.

If we denote by M the multisets, and V_A and V_B denotes the two possible notions of sets arising from it, following Variant A and Variant B respectively, we can draw the following diagram of the situation.



1.2 Outline

In Section 2 and 3 we set up a bit of framework to work within. Starting from Section 4 we will follow the path of Variant A.

Section 5 will take us through the basic lemmas about the type of iterative sets we define in Section 4. These lemmas are applied in Section 6 to give proofs that our model satisfies the axioms of Myhill's Constructive Set Theory.

In Section 7 we consider the problem of interpreting the two collection axioms of Aczel's CZF in our model.

In Section 8 we return to Variant B, which will make the relationship with the approach taken in Chapter 10 of the book "Homotopy Type Theory"[19] clear.

1.3 Notation

In what follows we will adhere to the following notation. Some of our notation is similar to that of the book "Homotopy Type Theory"[20], while some of it is inspired by the syntax of Agda.

[19]Ibid.
[20]Ibid.

- Function application will be denoted by juxtaposition, as in $f\,a$. Also, application of functions equipped with extra structure, such as equivalences and embeddings, will be denoted by juxtaposition.
- Quantifiiers, such as $\forall$,$\exists$,$\prod$ and $\sum$ bind weakly. For instance, $\prod_{x:\mathcal{M}} x \in a \to \sum_{y:\mathcal{M}} y \in B \wedge P\,x\,y$ disambiguates to $\prod_{x:\mathcal{M}} \left(x \in a \to \sum_{y:\mathcal{M}} y \in B \wedge P\,x\,y\right)$
- The equality sign, $=$, denotes the identity type. We sometimes equip it with a subscript emphasising which type the elements belong to, as in $a =_A a'$.
- Definitions are signified by $:=$.
- Judgemental equalities are denoted by $\equiv$.
- The notation A : Type denotes that A is a type,
- The notation A : Set denotes that A is a type which is a mere set and
- The notation A : Prop denotes that A is a type which is a mere proposition.
- The type U is a universe, with decoding family $T : U \to$ Type. We assume that this universe
  - is univalent,
  - contains the empty type, 0,
  - is closed under Π-types,
  - is closed under Σ-types,
  - is closed under $+$-types,
  - is closed under (-1)-truncation and
  - is closed under taking quotients of mere sets by equivalence relations.
- The notation e_A denotes the empty function $e : 0 \to A$. The subscript is dropped when inferable from the context.
- The notation $ap\,f$ refers to the usual function ap $f : a = a' \to f\,a = f\,a'$

2 Types and propositions

One of the main features of Martin-Löf's type theory is the interpretation of propositions as types[21]. The presence of the identity type gives the possibility of asking whether two proofs of a proposition are equal. A

[21] Martin-Löf 1984.

type in which all elements are equal is called, in homotopy type theory, a *mere proposition*.

The traditional interpretation of the existential quantifier in type theory is by the sigma type $\sum_{a:A} P\,a$. A proof of an existential proposition is thus a, p — a term $a : A$ of the quantified domain paired with a proof $p : P\,a$ that the term has the correct property. It is clear that since the existence is not necessarily unique, the type of such proof need not be a mere proposition.

In homotopy type theory, one introduces a truncated existential quantifier $\exists(x : A)P\,a$, which is constructed from $\sum_{a:A} P\,a$ by adding identifications of all elements in to make it a mere proposition. This gives the following introduction and elimination rule:[22]

$$\frac{\texttt{a : A} \qquad \texttt{b : B a}}{\texttt{[a,b] : } \exists\texttt{(a:A)(B a)}}\ \exists\texttt{-intro}$$

$$\frac{\texttt{x : } \exists\texttt{(a:A)(B a)} \qquad \texttt{y : } \exists\texttt{(a:A)(B a)}}{\texttt{q : x = y}}\ \exists\texttt{-quot}$$

$$\frac{\begin{array}{l}\texttt{P : } \exists\texttt{(a:A)(Ba)} \to \texttt{Prop} \\ \texttt{p : (a : A)} \to \texttt{(b : B a)} \to \texttt{P [a,b]}\end{array}}{\exists\texttt{-elim P p : (x : } \exists\texttt{(a:A)(B a))} \to \texttt{P x}}\ \exists\texttt{-elim}$$

Clearly $\sum_{a:A}(B\,a) \to \exists(a : A)(Ba)$ holds for all A and B. The opposite implication holds if $\sum_{a:A}(B\,a)$ is a mere proposition – which is to say that the existence is unique, with a unique proof.

The situation is similar for disjunctions. Traditionally, disjunction is interpreted as disjoint union in Martin-Löf's type theory, while homotopy type theory introduces a truncated variant. We will denote disjoint unions by the operator $+$ and the truncated disjunction by the operator $\vee$.

3 Models where equality is identity

In this section we define what an $\in$-structure is, and give some basic results on such structures in generality. We do this in order to adjust

[22] In these rules, written in an Agda-like notation, `Prop` refers to the type of mere propositions.

our expectation for the concrete model which will be main focus of this work.

3.1 $\in$-structures

Definition B:2. An *$\in$-structure* is a pair $(\mathcal{M}, \in)$ where $\mathcal{M} : \mathrm{Set}$ is a mere set, and $\in\colon \mathcal{M} \to \mathcal{M} \to \mathrm{Prop}$.

Definition B:3. For any $\in$-structure $(\mathcal{M}, \in)$ and element $a : \mathcal{M}$, we define $E\,a := \sum_{x:\mathcal{M}} x \in a$.

Definition B:4. An $\in$-structure $(\mathcal{M}, \in)$ is called *U-like* if for each $a : \mathcal{M}$ the type $E\,a$ is essentially U-small. That is, if each $a : \mathcal{M}$ the type $E\,a$ has a code in U.

Remark B:5. An $\in$-structure is basically a mere set with a merely propositional, binary predicate defined on it. The natural equality to consider for elements of such a structure is the identity type. This is in contrast to the setoid approach taken by Aczel.

"U-like" is meant to mimic the traditional terminology, "set-like", used in set theory.

3.2 Translations of first-order logic into type theory

Definition B:6. Given an $\in$-structure $(\mathcal{M}, \in)$ define two translations of formulas of first-order logic to type theory, $\sigma_{\mathcal{M},\in}$ and $\tau_{\mathcal{M},\in}$, by recursion on formulas. We let Context denote contexts (finite lists of variables, Variable Γ denoting the variables in Γ) and let Formula Γ denote formulas in a given context Γ : Context.

Starting with $\sigma_{\mathcal{M},\in}$, leaving out the subscripts for ease of reading:

$$\sigma : \prod_{\Gamma:\text{Context}} \text{Formula } \Gamma \to (\text{Variable } \Gamma \to \mathcal{M}) \to \text{Type}$$
$$\sigma\,\Gamma\,(\forall x\ \phi)\,\gamma := \prod_{a:\mathcal{M}} \sigma\,(\Gamma.x)\phi\,(\gamma.a)$$
$$\sigma\,\Gamma\,(\exists x\ \phi)\,\gamma := \sum_{a:\mathcal{M}} \sigma\,(\Gamma.x)\phi\,(\gamma.a)$$
$$\sigma\,\Gamma\,(\phi \wedge \psi)\,\gamma := \sigma\,\Gamma\,\phi\,\gamma \times \sigma\,\Gamma\,\psi\,\gamma$$
$$\sigma\,\Gamma\,(\phi \vee \psi)\,\gamma := \sigma\,\Gamma\,\phi\,\gamma + \sigma\,\Gamma\,\psi\,\gamma$$
$$\sigma\,\Gamma\,(\phi \to \psi)\,\gamma := \sigma\,\Gamma\,\phi\,\gamma \to \sigma\,\Gamma\,\psi\,\gamma$$
$$\sigma\,\Gamma\,\bot\,\gamma := 0$$
$$\sigma\,\Gamma\,\top\,\gamma := 1$$
$$\sigma\,\Gamma\,(x \in y)\,\gamma := \gamma\,x \in \gamma\,y$$
$$\sigma\,\Gamma\,(x = y)\,\gamma := \gamma\,x =_{\mathcal{M}} \gamma\,y$$

where $\Gamma.x$ denotes the context Γ extended with the variable x, and $\gamma.a$ denotes the function $\text{Variable}\,(\Gamma.x) \to \mathcal{M}$ which maps x to a.

We define $\tau_{\mathcal{M},\in}$ analogously, only difference being in the clauses for $\vee$ and $\exists$:

$$\tau : \prod_{\Gamma:\text{Context}} \text{Formula } \Gamma \to (\text{Variable } \Gamma \to \mathcal{M}) \to \text{Type}$$
$$\tau\,\Gamma\,(\forall x\ \phi)\,\gamma := \prod_{a:\mathcal{M}} \tau\,(\Gamma.x)\phi\,(\gamma.a)$$
$$\tau\,\Gamma\,(\exists x\ \phi)\,\gamma := \exists(a:\mathcal{M})\ \tau\,(\Gamma.x)\phi\,(\gamma.a)$$
$$\tau\,\Gamma\,(\phi \wedge \psi)\,\gamma := \tau\,\Gamma\,\phi\,\gamma \times \tau\,\Gamma\,\psi\,\gamma$$
$$\tau\,\Gamma\,(\phi \vee \psi)\,\gamma := \tau\,\Gamma\,\phi\,\gamma \vee \tau\,\Gamma\,\psi\,\gamma$$
$$\tau\,\Gamma\,(\phi \to \psi)\,\gamma := \tau\,\Gamma\,\phi\,\gamma \to \tau\,\Gamma\,\psi\,\gamma$$
$$\tau\,\Gamma\,\bot\,\gamma := 0$$
$$\tau\,\Gamma\,\top\,\gamma := 1$$
$$\tau\,\Gamma\,(x \in y)\,\gamma := \gamma\,x \in \gamma\,y$$
$$\tau\,\Gamma\,(x = y)\,\gamma := \gamma\,x =_{\mathcal{M}} \gamma\,y$$

Example B:7. The axiom of union is translated differently by the two translations:

$$\begin{aligned}
\text{UNION} :=& \forall x\, \exists u\, \forall z\ z \in u \leftrightarrow \exists y \in x\, z \in y \\
\sigma\,()\,(\text{UNION}) =& \sum_{u:\mathcal{M}} \prod_{z:\mathcal{M}} z \in u \leftrightarrow \sum_{y:\mathcal{M}} y \in x \wedge z \in y \\
\tau\,()\,(\text{UNION}) =& \exists (u : \mathcal{M})\ \prod_{z:\mathcal{M}} z \in u \leftrightarrow \exists (y : \mathcal{M})\ y \in x \wedge z \in y
\end{aligned}$$

If the structure $(\mathcal{M}, \in)$ satisfies the extensionality axiom, then the property $\prod_{z:\mathcal{M}} z \in u \leftrightarrow \exists (y : \mathcal{M})\ y \in x \wedge z \in y$ completely characterises u, making $\tau\,()\,(\text{UNION}) \simeq \sum_{u:\mathcal{M}} \prod_{z:\mathcal{M}} z \in u \leftrightarrow \exists (y : \mathcal{M})\ y \in x \wedge z \in y$. However, $\sum_{y:\mathcal{M}} y \in x \wedge z \in y$ is not always implied by $\exists (y : \mathcal{M})\ y \in x \wedge z \in y$ so the two axioms remain distinct.

3.3 Axioms of set theory

The axioms of set theory contain a number of axiom schemas, such as collection or replacement, or (restricted) separation. In set theory this adds one axiom for each first-order formula. In type theory it is more convenient to use the higher order features of type theory and regard these as quantified over all predicates. The result is much stronger than the original axiom scheme. In the following we will exploit this extra strength.

Definition B:8. Given a structure, $(\mathcal{M}, \in)$, and a predicate $P : \mathcal{M} \to \mathcal{M} \to \text{Type}$ and an element $m : \mathcal{M}$, we define *σ-replacement for P and m in* $(\mathcal{M}, \in)$ to be the type $\sigma_P\,(a)\,((\forall x \in a\, \exists! y\ P\,x\,y) \to \exists b\, \forall y (y \in b \leftrightarrow \exists x \in a\ P\,x\,y))\,m$, where σ_P is σ extended with the clause $\sigma\,\Gamma\,(P\,x\,y)\,\gamma := P\,(\gamma\,x)\,(\gamma\,y)$, in order to interpret the P as a predicate symbol.

Define τ-replacement in the same way, substituting τ for σ.

Proposition B:9. If an $\in$-structure $(\mathcal{M}, \in)$ satisfies extensionality, σ-replacement and has an ordered pairing operation $\langle -, - \rangle : \mathcal{M} \to \mathcal{M} \to \mathcal{M}$, then the following choice principle, stemming from the so-called *type theoretic choice principle*[23], holds:

For any $P : \mathcal{M} \to \mathcal{M} \to \text{Type}$ and $a, b : \mathcal{M}$ if

[23]The type theoretic choice principle is the fact that for all A, B and P, the type $(\prod_{a:A} \sum_{b:B\,a} P\,a\,b) \to \sum_{f:\prod_{a:A} B\,a} \prod_{a:A} P\,a\,(f\,a)$ is inhabited.

$$\prod_{x:\mathcal{M}} x \in a \to \sum_{y:\mathcal{M}} y \in B \wedge P\,x\,y,$$

then there is a function $f : \mathcal{M}$ with domain a and codomain b such that $\prod_{x:\mathcal{M}} P\,x\,(f\,x)$.

Proof: Given a,b and a proof $p : \prod_{x:\mathcal{M}} x \in a \to \sum_{y:\mathcal{M}} y \in B \wedge P\,x\,y$, apply σ-replacement to a and the predicate $P' : \mathcal{M} \to \mathcal{M} \to \mathrm{Prop}$ defined by

$$P'\,x\,z := \sum_{q:x\in a} z =_{\mathcal{M}} \langle x, \pi_0\,(p\,x\,q)\rangle$$

The resulting element of $\mathcal{M}$ is a function with the desired properties. □

Remark B:10. Proposition B:9 shows us that we cannot hope to have a model satisfying all axioms of constructive set theory by interpreting the existential quantifier as Σ-types while at the same time interpreting equality as the identity type of a mere set. This is because the above proposition shows that AC will hold, in this interpretation, and thus if all other CZF axioms hold (in fact U-restricted separation and pairing should suffice), Diaconescu's theorem demonstrates that the law of excluded middle holds (for all U-small propositions).

4 The model of iterative sets

We recall the definition, from Part A, of the type of iterated multisets and the membership relation.

Definition B:11.

$$M := W_U T$$

$$\in_M : M \to M \to \mathrm{Set}$$
$$x \in_M (\sup\, a\, f) := \sum_{i:T\,a} f\,i = x$$

The elements of M are the iterative multiset in which another element of M may occur any U-small number of times. This is expressed

by the relation $\in_M$, as follows: Given $x, y : M$ the type $x \in_M y$ is the type of occurrences of x in y. For instance, if $(x \in_M y) \simeq 2$ then x occurs twice in y.

What we would like to consider are the elements of M in which every element occurs at most once. Such a multiset would be called *set-like*. Being set-like could be expressed in different ways. The most direct would be to say that $y : M$ is set-like whenever $x \in_M y$ is a mere proposition for all other $x : M$. Another way to state this is to say that the function mapping each instance of an element in y to the element it represents is an embedding.

The *iterative sets* are those multisets which are hereditarily set-like. On M we define the predicate itset, recursively, as follows.

Definition B:12.

$$\begin{aligned} &\text{itset} : M \to \text{Type} \\ &\text{itset}\,(\sup\, a\, f) := \text{embedding}\ f \wedge \prod_{i:T\,a} \text{itset}\,(f\, a) \end{aligned}$$

For each $x : M$, whenever we have itset x, we say that x is an *iterative set*.

A Σ-type then collects the type of iterative sets as a subtype of M – with elementhood defined by the restriction of elementhood for multisets.

Definition B:13. The type of iterative sets, V, is defined as the subtype of M of iterative sets, or $V \equiv \sum_{x:M} \text{itset}\ x$.

We denote by $\in_V$ the specialisation of $\in_M$ to V, that is $x \in_V y := \pi_0\, x \in_M \pi_0\, y$.

Notation B:14. We will permit ourselves a slight abuse of notation when denoting elements of V. We will for the remainder of this article use the notation $\sup\, a\, f$ to denote an element of V constructed by $a : U$ and an embedding $f : T\, a \hookrightarrow V$.

Remark B:15. Checking that a multiset is an iterative set can be a tedious task, since it has to be carried out on each level, but the next section will give lemmas to make it easier to construct new iterative sets.

5 Basic results

In this section we clarify the basic properties of iterative multisets and $(V, \in_V)$. The most important of these is the extensionality of V with respect to $\in_V$.

We start with reminding ourselves of the extensionality theorem for multisets.

Theorem B:16. $x =_M y \simeq \prod_{z:M} ((z \in_M y) \simeq (z \in_M y))$

Proof: See Theorem A:14 in Subsection 3.5 of Part A.

Lemma B:17. The type itset x is a mere proposition for every $x : M$.

Proof: Immediate by induction on M and that embedding f is a mere proposition.

Lemma B:18. The function ap $\pi_0 : (x =_V y) \to (\pi_0\, x =_M \pi_0\, y)$ is an equivalence.

Proof: Simple consequence of itset being a mere proposition.

Lemma B:19. For any $x : M$ and $y : V$ we have that the type $x \in_M \pi_0 y$ is a mere proposition.

Proof: Assume $y \equiv (\sup\, a\, f, (p, q))$. Observe that $x \in_M \pi_0 y$ is the fibre of f over y. Since p proves f to be an embedding, and embeddings have propositional fibres, then $x \in_M \pi_0 y$ is a proposition. □

Lemma B:20.

$$\left(\prod_{z:V} (z \in_V x \leftrightarrow z \in_V y)\right) \simeq \left(\prod_{z:M} (z \in_M \pi_0 x \leftrightarrow z \in_M \pi_0 y)\right)$$

Proof: Both sides of the equivalence are mere propositions, so it is enough to show biimplication. Passing from right to left is trivial, so we show only the implication from left to right.

Assume $z : M$. If $z \in \pi_0 x$ then z must be an iterative set. Thus, $z \in y$ which is to say $z \in \pi_0 y$. This demonstrates $z \in \pi_0 x \to z \in \pi_0 y$. That $z \in \pi_0 y \to z \in \pi_0 x$ is shown symmetrically. Hence, $z \in \pi_0 x \leftrightarrow z \in \pi_0 y$, which completes the proof. □

Lemma B:21. Given a small type $a : U$ and a function $f : T\,a \to V$, there is a set image $a\,f$ such that

- for each $i : T\,a$ we have that $f\,i \in (\text{image}\,a\,f)$
- for any merely propositional predicate $P : V \to \text{Set}$, given $\prod_{i:T\,a} P\,(f\,i)$ we can prove that $\prod_{z:V}(z \in (\text{image}\,a\,f) \to P\,z)$.

Proof: Given $a : U$ and $f : T\,a \to V$. Take the image factorisation of f, which can be expressed as a simple higher inductive type. Since V is locally U-small, the image has a code $b : U$. Denote the injection of the image into V by $g : Tb \to V$, and define image $a\,f := \sup\,b\,v$. □

6 V models Myhill's constructive set theory

In this section we prove that $(V, \in_V)$ models the axioms of Myhill's Constructive Set Theory (CST), when the existential quantifiers are interpreted as truncated. In fact, we shall see that except for a few critical places, positive occurrences of the existential quantifier can be strengthened to $\sum$, mostly because the constructions we make are explicit.

6.1 Extensionality

The lemmas we have proved line up to give the following equivalence:

Extensionality B:22. $x =_V y \leftrightarrow \prod_{z:V} (z \in x \leftrightarrow z \in y)$

Proof:

$$\begin{aligned} x =_V y &\simeq \pi_0 x = \pi_0 y && (1)\\ &\simeq \prod_{z:M} (z \in_M \pi_0 x \simeq z \in_M \pi_0 y) && (2)\\ &\simeq \prod_{z:M} (z \in_M \pi_0 x \leftrightarrow z \in_M \pi_0 y) && (3)\\ &\simeq \prod_{z:V} (z \in_V \pi_0 x \leftrightarrow z \in_M \pi_0 y) && (4) \end{aligned}$$

□

6.2 The empty set, natural numbers

The empty set is given by $\emptyset := \sup 0e$. The natural numbers we constructed in Part A, Subsection 4.10 is indeed an iterative set.

6.3 Separation

Restricted separation can be done in V without quotienting, as long as the separating predicate is merely propositional. Thus, whenever we separate a formula with existential quantifier in a positive position, the existence must be unique or truncated. If there are more than one witness of the statement, the result would be a multiset where the element occurs once per witness of the statement. The same is true for disjunction. Notice that if $A + B$ is a proposition then A and B are mutually exclusive (since $l\,a$ and $r\,b$ are always distinct).

Proposition B:23. For any U-small predicate $P : Vi \to \mathrm{Prop}$, and $x : V$ there is $u : V$ such that for any $z : V$ we have $z \in u \leftrightarrow (P\,z) \times (z \in x)$.

Proof: Assume that $x \equiv \sup\, A\, f$ and let $u := \sup(\sum_{a:A} P\,(f\,a))(f \circ \pi_0)$. Since $P \circ f$ is a mere proposition $\pi_0 : (\sum_{a:A} P\,(f\,a)) \to A$ is injective, thus $f \circ \pi_0$ is injective.

If $z \in u$ then $z = f\,a$ for some $a : A$ such that $P\,(f\,a)$, and hence $P\,z$ and $z \in x$.

If $p : P\,z$ and $q : z \in x$ then let $a := \pi_0 q$, $((a, p), \pi_1\,q)$ will prove that $z \in u$. □

6.3.1 $\in$-induction

Induction on V can be performed, even when the predicate is a general type, not necessarily merely propositional.

Proposition B:24. For every predicate $P : V \to \mathrm{Type}$, if for each $x : V$ we have that $\prod_{y:V} y \in x \to P\,y$ implies $P\,x$, then we have $P\,x$ for every $x : V$.

Proof. By induction on V. Assume $x \equiv \sup\, A\, f$ then by induction hypothesis $P\,(f\,i)$ for every $i : A$. We must show that if $y \in x$ then $P\,y$. However, $y \in x$ means that there is i such that $y = f\,i$ so we can transport the induction hypothesis to obtain $P\,y$. Thus, we have shown $\prod_{y:V} y \in x \to P\,y$, which implies $P\,x$ by assumption. □

6.4 Pairing and Union

In order to show pairing and union we will have to apply the image construction, previously introduced (Lemma B:21). If we applied the constructions of union and pairing for multisets which we defined in

Part A, then the resulting multisets would not be set-like. Therefore, we need a different construction, and the most natural is to take quotients to make the multisets back into sets. The fact that quotienting was not needed for multisets, may be an indication that iterative multisets is a more natural notion than iterative sets to consider in type theory.

Definition B:25. Given $x, y : V$, we define $\{x, y\} := \text{image}\,(1 + 1)\,(\text{const}\,x + \text{const}\,y)$.

Proposition B:26. For any $x, y : V$ and for each $z : V$ we have that $z \in \{x, y\} \leftrightarrow ((x = z) \vee (y = z))$

Proof: Simple consequence of Lemma B:21. □

Definition B:27. Given $x : V$, where $x \equiv \sup\, A\, f$ define

$\cup x := \text{image}\,(\sum_{a:A}(\overline{f\,a}))\,(\lambda\langle i, j\rangle.\widetilde{(f\,i)}\,j)$.

Proposition B:28. For any $x : V$ and for any $z : V$, we have that $z \in \cup x \leftrightarrow \exists y\ (y \in x \wedge z \in y)$

Proof: Simple consequence of Lemma B:21. □

Remark B:29. The truncated existential quantifier in the above proposition cannot be strengthened to a Σ-type, since it would mean constructing sections for almost arbitrary quotients.

6.5 Replacement

Proposition B:30. For any $a : V$ and $P : V \to V \to \text{Prop}$, such that for all $x \in a$ there exists a unique y for which $P\,x\,y$, then there is $b : V$ such that for each $y : V$ we have that $y \in b$ if and only if there exists $x \in a$ such that $P\,x\,y$.

Proof: Given $a = \sup\,\bar{a}\,\tilde{a}$, the assumptions let us construct a map $T\,\bar{a} \to V$. Using Lemma B:21, we can construct its image in V, which will have the desired properties. □

Proposition B:31. For any $a : V$ and $F : V \to V$, there is $y : V$ such that for any $z : V$ we have $z \in b \leftrightarrow \exists(w : V)\ z = F\,w$.

Proof: Assume $x \equiv \sup\, a\, f$ and let $b = \text{image}\,(F \circ f)$, and apply Lemma B:21. □

6.6 Exponentials

The construction of exponentials in this model is particularly easy, since a set-theoretical function between two elements, $(\sup\, A\, f)$ and $(\sup\, B\, g)$, of V boils down to a function in the type theory $A \to B$. Instead of giving a direct proof, we can lean on Part A, Subsection 4.9 in order to prove the correctness of this construction.

6.6.1 Exponentiation

Exponentiation of sets is a special case of exponentiation of multisets. For multisets, we defined[24] $\mathrm{operation}_{a,b}\ f$ for every $a, b, f : M$, and showed that there is a multiset $\mathrm{Exp}\ a\, b$, such that $f \in \mathrm{Exp}\ a\, b \simeq \mathrm{operation}_{a,b}\ f$. We will here show that whenever $a, b, f : V$, we have $\mathrm{operation}_{\pi_0\, a, \pi_0\, b}\, (\pi_0\, f)$ if an only if $\mathrm{Fun}\ a\, b\, f$, and that in fact itset $(\mathrm{Exp}\, (\pi_0\, a)\, (\pi_0\, b))$, in order to conclude that there is exponentiation in V.

Lemma B:32. Whenever $a, b, f : V$, we have that $\mathrm{operation}_{\pi_0\, a, \pi_0\, b}\, (\pi_0\, f) \leftrightarrow \mathrm{Fun}\ a\, b\, f$.

Proof: Recall that[25] for every $a, b, f : M$,

$$\begin{aligned}\mathrm{operation}_{a\, b}\ f := &\left(\prod_z z \in f \to \sum_x \sum_y z = \langle x, y\rangle\right)\\ &\wedge \left(\prod_x x \in a \simeq \sum_y \langle x, y\rangle \in f\right)\\ &\wedge \left(\prod_y y \in b \leftarrow \sum_x \langle x, y\rangle \in f\right)\end{aligned}$$

If here it said $\leftrightarrow$ instead of $\simeq$, this would state that f is a total relation between a and b. We thus have to show that this strengthening is exactly the same as ensuring functionality of a total relation.

In the middle conjunction, $x \in a$ is a mere proposition since a is an iterative set by assumption. Thus, $\sum_y \langle x, y\rangle \in f$ must also be a mere proposition, and thus equivalent to $\exists!(y : M)\langle x, y\rangle \in f$. This is exactly the requirement for a total relation to be functional. □

Remark: Since $\mathrm{operation}_{\pi_0\, a, \pi_0\, b}\, (\pi_0\, f)$ and $\mathrm{Fun}\ a\, b\, f$ are both mere propositions, the biimplication is also an equivalence of types.

---

[24] Definition A:24 in Subsection 4.7 of Part A

[25] ibid.

Lemma B:33. For every $a, b : V$ the multiset $\mathrm{Exp}\,(\pi_0\,a)\,(\pi_0\,b)$ is an iterative set.

Proof: The definition of $\mathrm{Exp}\,a\,(\pi_0 b)$ is...

$$\begin{aligned}\mathrm{Exp}\,(\pi_0 a)\,(\pi_0 b) := \sup\,(\overline{(\pi_0\,a)} \to \overline{(\pi_0\,b)})\\ (\lambda f.\,\sup\,\overline{(\pi_0\,a)}\\ (\lambda i.\langle\widetilde{(\pi_0\,a)}i, \widetilde{(\pi_0\,b)}\,(f\,i)\rangle))\end{aligned}$$

Thus, we need to show that the function $\lambda f.\,\sup\,\overline{(\pi_0\,a)}\,(\lambda i.\langle\widetilde{(\pi_0\,a)}i, \widetilde{(\pi_0\,b)}(f\,i)\rangle)$, which takes a function to its graph, is injective, but this comes down to that a function is determined by its graph, which follows from function extensionality.

Next, we need to argue that each graph of each function is itself an iterative set. Assume that $f : \overline{(\pi_0\,a)} \to \overline{(\pi_0\,b)}$, then the function which maps an element $i : \overline{(\pi_0\,a)}$ to $\langle\widetilde{(\pi_0\,a)}\,i, \widetilde{(\pi_0\,b)}\,(f i)\rangle$ is injective since $\pi_0\,a$ is an iterative set. Furthermore, $\langle\widetilde{(\pi_0\,a)}\,i, \widetilde{(\pi_0\,b)}\,(f\,i)\rangle$ is an iterative set since both a and b are iterative sets. Thus, the graph of f is an iterative set. □

Proposition B:34. For every $a, b : V$ there is a $u : V$ such that for any $c : V$ there is an biimplication $c \in u \leftrightarrow \mathrm{Fun}\,a\,b\,c$.

Proof: Direct from Lemma B:32, Lemma B:33, and exponentiation in M[26], letting $u := \mathrm{Exp}\,(\pi_0 a)\,(\pi_0 b) \leftrightarrow \mathrm{Fun}\,a\,b\,f$. □

7 Collection axioms

In this section we discuss the status of collection axioms of constructive set theory in our model.

Neither strong collection, nor subset collection seem to hold in either extreme interpretation of the existential quantifier (i.e. applying τ or σ). Interpreting the existential quantifier as Σ-types forces us to make arbitrary choices in an apparently unconstructive way. Interpreting the existential quantifier as the truncated $\exists$, the assumptions become too weak to work with.

We discuss two approaches to solving this problem. On the one hand, we identify which existential quantifiers need to be weakened –

[26] `(M-EXP)` in Subsection 4.9, Part A

and which have to remain strict – in order to get something like the collection axioms to become provable for our model. On the other hand, we identify axioms about the type theoretical universe, from which V was constructed, from which we can derive collection and subset collection in the truncated form.

7.1 Strong Collection

Together with subset collection, strong collection is the most subtle of the axioms of constructive set theory. First of all because it is underspecified: the strong collection axiom states the existence of a set, but does not define it up to equality.

On an intuitive level, strong collection states that if we have shown that for all elements of a set x there exists some element with a certain property, then the proof is in some way an operation which should have an image in V.

The problem is that the proof in first-order logic may not be uniform, in the sense that the operation may not respect equality of elements. However, this would not prevent us from taking its image.

In Aczel's original model one can see this, as his V is a setoid and we can talk about extensional operations on the underlying type. However, our V has the identity type as its equality type. This means that if we interpret the existential quantifier as the Σ-type, then any proof operation will give rise to an actual function which respects equality. The image of such an operation is naturally a multiset, and we have seen how to quotient such a multiset to a set. The only wrinkle of this approach is that the "strong part" of strong collection has to be weakened by using a truncated existential quantifier.

7.1.1 Strong quantifiers

The following proposition is the rendering of strong collection in which we, as far as our abilities go, interpreted the existential quantifier as Σ-types.

Proposition B:35. For any predicate $P : V \to V \to \mathrm{Type}$ and every $a : V$ if $\prod_{x:V}(x \in a) \to \sum_{y:V} P\,x\,y$ then there is $b : V$ such that

1. $\prod_{x:V}(x \in a) \to \sum_{y:V} y \in b \wedge (P\,x\,y)$
2. $\prod_{y:V}(y \in b) \to \exists(x : V)\ x \in a \wedge (P\,x\,y)$

Proof: Given $a \equiv \sup\, \bar{a}\, \tilde{a}$, the assumptions let us construct a map $T\,\bar{a} \to V$. Using Lemma B:21, we can construct its image in V, which will have the desired properties. □

Remark B:36. It is well known that, in constructive set theory, collection implies replacement, and that the converse implication does not hold. But the strong version of replacement proven for our model in Proposition B:30, formulated for a given $\in$-structure, does in fact entail the collection principle of Proposition B:35, for the same $\in$-structure.

7.1.2 Weak quantifies

Aczel and Gambino (2006) discuss a general approach to interpreting first-order logic into type theory. For any given interpretation in their framework, they identify type theoretic principles corresponding to strong collection and subset collection. Here we will, in a similar fashion, give a sufficient principle for our model to satisfy strong collection in the sense of weak quantifiers.

Definition B:37. Collection Principle for a universe, U:
For every locally U-small $B : \mathrm{Set}$, given $a : U$ and $P : T\,a \to B \to \mathrm{Type}$, such that $\prod_{x:T\,A} \exists(y : B) P\,x\,y$ then $\exists(r : U)\exists(\beta : T\,r \hookrightarrow B)(\prod_{x:T\,a} \exists(y : T\,r) P\,x\,(\beta\,y)) \wedge (\prod_{y:T\,r} \exists(x : T\,a) P\,x\,y)$.

Proposition B:38. The collection principle for U implies strong collection in the following sense:
For any predicate $P : V \to V \to \mathrm{Type}$ and every $a : V$ if $\prod_{x:V}(x \in a) \to \exists(y : V) P\,x\,y$ then there merely exists $b : V$ such that

1. $\prod_{x:V}(x \in a) \to \exists(y : V)\ y \in b \wedge (P\,x\,y)$
2. $\prod_{y:V}(y \in b) \to \exists(x : V)\ x \in a \wedge (P\,x\,y)$

Proof: Apply the collection principle for U to $P \circ \bar{a}$, to obtain the mere existence r and β, which together form $b := \sup\, r\,\beta$. 1. and 2. follow from the property of the collection principle for U. □

7.2 Subset Collection

Subset collection is the principle that for any pair of sets there is a third set, such that each total relation between the two has an image

in the third. The precise formulation of the axiom in first-order logic is slightly complicated by the fact that one cannot quantify over all (or even all definable) total relations in the middle of a formula. Therefore, the axiom is an axiom schema quantified over ternary formulas, where the first parameter is allowed to vary. Our formulation will be close to the first-order formulation:

$$\begin{aligned}\forall a, b\ \exists c\ \forall u\ (\forall x \in a \to \exists y \in b \wedge P\,u\,x\,y) \\ \to (\exists d \in c \wedge (\forall x \in a \to \exists y \in d \wedge P\,u\,x\,y) \\ \wedge (\forall y \in d \to \exists x\ x \in a \wedge P\,u\,x\,y)\end{aligned}$$

In the above formula there are three existential quantifiers to interpret, either as a Σ-type or as a truncated existential. At one extreme it is possible to give all but the last existential quantifier the Σ-type interpretation, truncating the last one.

Proposition B:39. For every predicate $P : V \to V \to V \to \mathrm{Type}$ and every $a, b : V$ there is $c : V$ such that for all $u : V$ if $\prod_{x:V}(x \in a) \to \sum_{y:V} P\,u\,x\,y$ then there is $d \in c$ such that

1. $\prod_{x:V}(x \in a) \to \sum_{y:V} y \in d \wedge (P\,u\,x\,y)$
2. $\prod_{y:V}(y \in d) \to \exists(x : V)\ x \in a \wedge (P\,u\,x\,y)$

Proof: Given $a \equiv \sup\,\bar{a}\,\tilde{a}$ and $b \equiv \sup\,\bar{b}\,\tilde{b}$, consider the function $\gamma : (\bar{a} \to \bar{b}) \to V$ which maps each $f : \bar{a} \to \bar{b}$ to $\mathrm{image}(\tilde{b} \circ f)$ in V. Let $c :=$ image γ, which will have the desired properties. □

7.2.1 Weak quantifiers

In the same way we did for strong collection, we identify a principle of subset collection for our universe which is sufficient to prove the truncated variation of subset collection for our model.

Definition B:40. Subset Collection Principle for U:

For each $a, b : U$ there merely exists $c : U$ such that for every $P : T\,a \to T\,b \to \mathrm{Type}$ such that $\prod_{x:T\,a} \exists(y : T\,b) P\,x\,y$, there $\exists(r : T\,c)(\prod_{x:T\,a} \exists(y : T\,r) P\,x\,(\alpha\,y)) \wedge (\prod_{y:T\,r} \exists(x : T\,a) P\,x\,y)$

Proposition B:41. For every predicate $P : V \to V \to V \to \text{Type}$ and every $a, b : V$, there merely exists a $c : V$ such that for every $u : V$, if $\prod_{x:V}(x \in a) \to \sum_{y:V} P\,u\,x\,y$ then there merely exists $d \in c$ such that

1. $\prod_{x:V}(x \in a) \to \exists(y : V)\ y \in d \wedge (P\,u\,x\,y)$
2. $\prod_{y:V}(y \in d) \to \exists(x : V)\ x \in a \wedge (P\,u\,x\,y)$

Proof: Apply the subset collection principle for U to a,b to obtain the mere existence of c, and then use the property of c on the predicate $P\,u$.

8 Equivalence with the HIT-formulation

Section 5.1 of Chapter 10 of the book "Homotopy Type Theory"[27], is dedicated to set theory in the context of homotopy type theory. The iterative hierarchy is presented there, in the form of a higher inductive type (HIT). Our V is a HIT-free alternative to this, and in this section we show that the two types are equivalent.

The HIT formulation bears more than a slight similarity to Aczel's original construction of the iterative hierarchy in type theory. Both are quotients of the type we call M. The difference is that while Aczel uses the untruncated version of the quantifiers and leaves the quotient a setoid, HTT uses the truncated quantifiers and postulates the existence of a type with the identity type to match.

In this section we define the bisimulation relation on M of which the V in HTT Chapter 10 can be seen a quotient. We show that this quotient is equivalent to our V. The proof on this relies on a function, iterative-image : $M \to V$, which turns any multiset into an iterative set by identifying all occurrences of each elements, having first applied the process inductively on all elements.

Definition B:42. We define by induction on M the binary relation:

$$\approx : M \to M \to \text{Type}$$

$$(\sup\ a\,f) \approx (\sup\ b\,g) := \left(\prod_{x:T\,a} \exists(y : T\,b)(f\,x \approx g\,y)\right) \times \left(\prod_{y:T\,b} \exists(x : T\,a)(f\,x \approx g\,y)\right)$$

[27] Univalent Foundations Program 2013.

Definition B:43. We define by induction on M the function:

$$\begin{aligned}&\text{iterative-image} : M \to V\\ &\text{iterative-image}\,(\sup\, a\, f) := \text{image}\, a\, (\text{iterative-image} \circ f)\end{aligned}$$

Lemma B:44. For each $x : M$ we have that $x \approx \text{iterative-image}\ x$.

Proof: For each element of x there is a corresponding element in iterative-image x, since the type of elements of iterative-image x is a quotient of the type of elements in x. In the other direction there just *merely* exists an element of x for each element of iterative-image x, since it was a quotient. However, this is sufficient to show that $x \approx$ iterative-image x.

Lemma B:45. For iterative sets $\approx$ is equivalent to the identity type. That is, for every $x, y : M$ given itset x and itset y, we have that $x \approx y \to x =_M y$.

Proof: By W-induction. Let $x \equiv \sup\, a\, f$ and $y \equiv \sup\, b\, g$, and assume $x \approx y$. By extensionality, it suffices to show that for any $z : M$ we have that $z \in_M x$ if and only if $z \in_M y$.

In one direction, if $z \in_M x$ we know that there is $i : T\, a$ such that $z =_M f\, i$. Since $z \in_M y$ is a mere proposition, we can assume from $x \approx y$ that there is a $j : T\, b$ such that $f\, i \approx g\, j$. By inductive hypothesis, $f\, i \approx g\, j \to f\, i =_M g\, j$, and thus $z =_M g\, j$ which is to say $z \in_M y$.

The other direction is symmetric in x and y.

Proposition B:46. $V \simeq M/\approx$

Proof: Direct consequence of the previous two lemmas.

Remark B:47. That $M/\approx$ is equivalent to the HIT formulation in the book can be seen from Lemma 10.5.5 of "Homotopy Type Theory"[28]. Thus, proposition B:46 shows that our V is indeed equivalent to the HIT formulation of V.

[28] Ibid.

References

Aczel, Peter (1978). "The Type Theoretic Interpretation of Constructive Set Theory". In: *Logic Colloquium '77*. Ed. by A. MacIntyre, L. Pacholski, and J. Paris. North–Holland, Amsterdam-New York, pp. 55–66.

Aczel, Peter and Nicola Gambino (2006). "The generalised type-theoretic interpretation of constructive set theory". In: *J. Symbolic Logic* 71.1, pp. 67–103. DOI: 10.2178/jsl/1140641163. URL: http://dx.doi.org/10.2178/jsl/1140641163.

Martin-Löf, Per (1984). *Intuitionistic type theory. Notes by Giovanni Sambin*. Vol. 1. Studies in Proof Theory. Bibliopolis, Naples, pp. iv+91. ISBN: 88-7088-105-9.

Univalent Foundations Program, The (2013). *Homotopy Type Theory: Univalent Foundations of Mathematics*. Institute for Advanced Study: homotopytypetheory.org. URL: http://homotopytypetheory.org/book.

C

Agda Formalisation

Abstract

The following is a formalisation of many of the proofs of Part A and Part B in Agda. Agda is a proof-checker and programming language based on dependent type theory and pattern matching, which allow a very close correspondence between the style of proofs in the article and the formalised, machine-readable proof.

Some proofs are present both in the article and the formalisation, while others are only present in one of them. Some proofs in the article are mere sketches, while the details are in the formalised proof.

The original motivation for starting this formalisation was to make the details of Lemma A:13 in Part A clear. In fact, the informal proof in the article only reached its current form after the formal proof was completed. After the formalised proof was done it was a lot easier to pick out the important elements, which a human might be interested in, hiding the tedious details behind the phrase "... can be verified by Id-induction...".

A strength of Agda is that it gives easy access to term-normalisation. Being able to normalise terms quickly opens up for more experimentation to simplify proofs than if each normalisation would be carried out by hand. In our development this has manifested as simplifications of the proof of Lemma A:6 and Lemma A:12 of Part A. Both of which were first proved by more complicated path computations, but once formalised, inserting the reflexivity proof various places, `idp`, and allowing Agda to attempt unification, showed that there were simplifications to be made.

1 Prerequisites

Proof-checkers are pieces of software, undergoing development for long periods of time and therefore existing in many versions. Agda is no exception, and it is a matter of fact that a proof formalised and checked by one version may fail to check by the next version of Agda. This may be due to introduction of errors in the new version, removal of errors

from the previous version, or due to intended changes in syntax. It is therefore of essence to specify along with any formalisation what version of the proof-checker the code has been checked with. If the formalisation depend on any library, such must also be specified with versions. In practise, locating and installing old versions of all dependencies – and all dependencies of dependencies, etc. – on a new system, may prove difficult.

An alternative route, which requires some dedication, is to keep the formalisation up to date with the latest changes in the proof-checker. At the very least, one should aim to write the formalisation in such a way that it is not only machine-readable, but also human-readable, so that in case the specific version of the proof-checker should disappear, and the code cease to verify on any available versions, the intention behind the code should be clear so that any able reader can themself perform the changes needed for the code to verify on a later version.

Our formalisation depends on the HoTT-Agda library, which is a development of Homotopy Type Theory in Agda.

Prerequisite	Tested versions	URL
Agda	2.4.2.4	`https://github.com/agda/agda/archive/2.4.2.4.tar.gz`
HoTT-agda	eb24ea20e1a28de31	`https://github.com/HoTT/HoTT-Agda/tree/eb24ea20e1a28de31`

2 Assumptions of the formalisation

Agda implements dependent type theory with pattern matching. Traditionally, the way pattern matching has been implemented allows proofs of the uniqueness of identity proofs (UIP) to type check[29], which is inconsistent with Univalence. However, recent version includes an option to tighten the restraints on pattern matching so that one can no-longer prove UIP[30]. This allows HoTT-agda, upon which our development depends, to postulate the Univalence Axiom.

Agda allows defining new recursive data types and functions. To ensure consistency of these it applies positivity checks and termination checks. However, all types used in our development are well-known from type theory literature, and thus we believe that the proofs should be

[29]The principle states that for $a, b : A$ and any $\alpha, \beta : \mathrm{Id}_A\, a\, b$ we have that $\mathrm{Id}\, \alpha\, \beta$.

[30]Cockx, Devriese, and Piessens 2014.

transferable to any proof-checker with a type system able to represent the following type constructs:

- Π-types
- Σ-types
- Binary sum types
- W-types
- Id-types
- A univalent universe
- (-1)-truncation (for set theoretical axioms on V)
- Mere set quotients (for set theoretical axioms on V)

The proofs concerning the identity type of W-types uses, η-reduction for functions (i.e. $(\lambda x.fx) \equiv f$). One may speculate that the proofs could be modified to hold even if η-reduction only holds up to identity, but most dependent type systems have some function type with η-reduction, so we do not pursue that road here.

3 Case study

To demonstrate how the human readable proofs and the formal proofs relate, we will have a closer look at a single theorem, and its incarnation in both article and code. The theorem chosen is the theorem which shows that our chosen notion of equality for multisets is equivalent to the identity type – given the univalence axiom. This is found in Part A as Theorem A:9. The code formalising the theorem is quoted below.

```
Id-is-Eq' : (i : ULevel) (x y : M i) → (x == y) ≃ Eq' x y
Id-is-Eq' i (sup A f) (sup B g) = IH ∘e EXT ∘e UA ∘e WLEM where

   IH : Σ (A ≃ B) (\α → ((x : A) → (g (apply α x)) == (f x)))
      ≃ Eq' (sup A f) (sup B g)
   IH = equiv-Σ-snd (\α → equiv-Π (ide _)
                                  (\x → Id-is-Eq' i (g (apply α x)) (f x)))

   EXT : Σ (A ≃ B) (\α → (g ∘ (apply α) == f))
       ≃ Σ (A ≃ B) (\α → ((x : A) → (g (apply α x)) == (f x)))
   EXT = equiv-Σ-snd (\α → app=-equiv)

   lem0 : {A : Type i} → (ua (ide A)) == idp
   lem0 = ua-η idp
   lem = equiv-induction
          (\α → transport (T i) (fst ua-equiv α) == apply α)
          (\A → ap (\f → coe (ap (T i) f)) lem0)
   UA : (Σ (A == B) (\α → (g ∘ (transport (T i) α) == f)))
      ≃  Σ (A ≃ B)  (\α → (g ∘ (apply α) == f))
   UA = (equiv-Σ-snd (\α → coe-equiv (ap (\h → g ∘ h == f) (lem α))))
     ∘e (equiv-Σ-fst (\α → g ∘ (transport (T i) α) == f)(snd ua-equiv)) ⁻¹
```

```
  WLEM : ((sup A f) == (sup B g))
       ≃ Σ (A == B) (\α → (g ∘ (transport (T i) α) == f))
  WLEM = W-id-≃ (T i)


Id≃Eq : (i : ULevel) (x y : M i) → (x == y) ≃ Eq x y
Id≃Eq = \i x y → Eq'-is-Eq x y ∘e Id-is-Eq' i x y
```

Starting at the bottom, the term `Id≃Eq` is the proof of the theorem we want. We can see the statement of the theorem on the line above. The type `Eq` is the name in the formalisation of $=_M$, and '==' is the identity type.

While M in our paper was defined with an implicit dependency of a universe U, the Agda formalisation defines `M i` having an explicit dependency of a universe level `i`. That's why the type of `Id≃Eq` starts with the quantification `(i : ULevel)`.

Looking at the definition of `Id≃Eq` one quickly sees mention of something called `Eq'`, which doesn't appear in the proof of the article. This is because in its natural formulation `Eq` lies one universe level below the identity type of `M`, and it turned out that the cleanest way to handle this in Agda was to define an auxiliary type with the same inductive definition as `Eq` on a higher level.

Thus, the essence of the proof is in the term `Id-is-Eq'`, and it closely corresponds to the informal proof, which is a chain of equivalences. In the formalisation each step is given short name: `IH`, `EXT`, `UA` and `WLEM`.

The informal proof has an addition step helping the reader recognise the definition of a homotopy between two function. Agda usually needs no help recognising definitional equalities. However, each step in the proof is easy enough that the reader will understand without further explanation, Agda needs to be presented the exact terms. Each step therefore is given a definition in the formalisation – in each case a simple adaption of some already known term. Perhaps the only step which seems unduly complicated is the definition of `UA`. This might be due to the fact that the univalence axiom is a kind of artificial addition to Agda, which is simply postulated and doesn't come with many convenient definitional equalities, forcing a bit of transport yoga.

In conclusion, we see that there is as close a correspondence between the informal proof and the Agda formalisation as we could hope for. However, they differ in which details we chose to leave to the reader/proof-checker. The human can be trusted to see how to apply already known theorems in the relevant way, while Agda excel at computing definitional equalities.

4 Selected excerps from HoTT-agda

The formalisation of our results depends on the HoTT-Agda library[31] – mostly for standard definitions of Homotopy Type Theory. In stead of quoting lengthy files of code from the library, we include here the types of some of the terms we use in our formalisation.

```
PathOver : ∀ {i j} {A : Type i} (B : A → Type j)
  {x y : A} (p : x == y) (u : B x) (v : B y) → Type j

syntax PathOver B p u v =
  u == v [ B ↓ p ]

ap : ∀ {i j} {A : Type i} {B : Type j} (f : A → B) {x y : A}
  → (x == y → f x == f y)

transport : ∀ {i j} {A : Type i} (B : A → Type j) {x y : A} (p : x == y)
  → (B x → B y)

Π : ∀ {i j} (A : Type i) (P : A → Type j) → Type (lmax i j)
Σ : ∀ {i j} (A : Type i) (B : A → Type j) → Type (lmax i j)

_•_ : {x y z : A}
    → (x == y → y == z → x == z)

_•'_ : {x y z : A}
     → (x == y → y == z → x == z)

•'=• : {x y z : A} (p : x == y) (q : y == z)
     → p •' q == p • q

•'-assoc : {x y z t : A} (p : x == y) (q : y == z) (r : z == t)
    → (p •' q) •' r == p •' (q •' r)

•-unit-r : {x y : A} (q : x == y) → q • idp == q

•'-unit-l : {x y : A} (q : x == y) → idp •' q == q

! : {x y : A} → (x == y → y == x)

!-inv-l : {x y : A} (p : x == y) → (! p) • p == idp
!-inv-r : {x y : A} (p : x == y) → p • (! p) == idp

 _•ᵈ_ : {x y z : A} {p : x == y} {p' : y == z}
        {u : B x} {v : B y} {w : B z}
      → (u == v [ B ↓ p ]
      → v == w [ B ↓ p' ]
      → u == w [ B ↓ (p • p') ])
```

[31] The HoTT-Agda library is © 2013 Guillaume Brunerie, Evan Cavallo, Favonia, Nicolai Kraus, Dan Licata, Christian Sattler

5 Code

5.1 Propositions/Equivalences.magda

This module contains well-known lemmas about biimplication, and a few other lemmas about propositions which might be more logical to put somewhere else.

```
module Propositions.Equivalences where

open import lib.Base
open import lib.NType
open import lib.NType2
open import lib.Equivalences
open import lib.types.Sigma
open import lib.types.Pi
```

5.1.1 Notation

Apply equivalences (the notation in lib.Equivalences is annoying)

```
apply = fst

infix 4 _↔_
```

5.1.2 Logical equivalence

```
_↔_ : ∀ {i j} → Type i → Type j → Type (lmax i j)
_↔_ = \a b → (a → b) × (b → a)
```

For propositions, $\leftrightarrow$ and $\simeq$ are equivalent:

```
↔-level : ∀ {i j} {n : ℕ₋₂} {A : Type i} {B : Type j}
  → (has-level n A → has-level n B → has-level n (A ↔ B))
↔-level Al Bl = ×-level (→-level Bl) (→-level Al)

is-prop-if-true : ∀ {i} {A : Type i} → (A → is-prop A) → is-prop A
is-prop-if-true p = \x → p x x

prop-≃-is-↔ : ∀ {i j} {A : Type i} {B : Type j}
            → is-prop A → is-prop B
            → (A ≃ B) ≃ (A ↔ B)
prop-≃-is-↔ p q = (\α → apply α , apply (α ⁻¹)) , is-eq _ ϕ κ δ where
   ϕ = \e → (fst e) , is-eq _ (snd e)
                            (\b → prop-has-all-paths q _ _)
                            (\a → prop-has-all-paths p _ _)
   κ = \e → prop-has-all-paths (↔-level p q) _ _
   δ = \e → prop-has-all-paths (≃-level p q) _ _
```

Show that Σ over a contractible predicate changes nothing to the underlying type.

```
prop-equal : ∀ {i j} {A : Type i}
           → {P : A → Type j}
           → (∀ a → is-prop (P a))
           → {a a' : A} {p : P a} {p' : P a'}
           → (a == a')
           → (a , p) == (a' , p')
prop-equal q idp = pair= idp (prop-has-all-paths (q _) _ _)

has-all-paths-over : ∀ {i j} {A : Type i}
           → (B : A → Type j) → Type (lmax j i)
has-all-paths-over {A = A} B
           = {a a' : A} (p : a == a')
           → (b : B a) (b' : B a')
           → b == b' [ B ↓ p ]

contr-has-all-paths-over : ∀ {i j} {A : Type i}
                         → {B : A → Type j}
                         → Π A (is-contr ∘ B) → has-all-paths-over B
contr-has-all-paths-over c idp b b' = contr-has-all-paths (c _) b b'

Σ-contract : ∀ {i j} {A : Type i}
           → {P : A → Type j}
           → (∀ a → is-contr (P a))
           → Σ A P ≃ A
Σ-contract c = fst , is-eq _ (\a → (a , fst (c a))) ϕ ψ where
           ϕ = \a → idp
           ψ = \ap → pair= idp (contr-has-all-paths-over c _ _ _)
```

Dependent equality in fibres of propositional predicates is contractible. This is the dependent version of the definition of a mere proposition.

```
is-prop-has-contr-path-over : ∀ {i j} {A : Type i}
           → {B : A → Type j}
           → (∀ a → is-prop (B a))
           → {a a' : A} → (p : a == a')
           → (b : B a) (b' : B a')
           → is-contr (b == b' [ B ↓ p ])
is-prop-has-contr-path-over c idp = c _
```

If a predicate is propositional, the first projection is an embedding.

```
prop-equal-≃ : ∀ {i j} {A : Type i}
           → {P : A → Type j}
           → (∀ a → is-prop (P a))
           → {a a' : A} (p : P a) (p' : P a')
           → (a == a') ≃ ((a , p) == (a' , p'))
prop-equal-≃ q p p' = =Σ-eqv (_ , p ) (_ , p')
                    ∘e (Σ-contract
                          (\p → is-prop-has-contr-path-over q p _ _) ⁻¹)
```

Sufficient criteria to establish equivalence between two Σ-types when the families are propositional predicates.

```
restricted-equiv : ∀ {i j k l} {A : Type i} {B : Type j}
                 → (α : A ≃ B)
```

```
                → {P : A → Type k} {Q : B → Type l}
                → (∀ a → is-prop (P a))
                → (∀ b → is-prop (Q b))
                → (∀ a → P a → Q (apply α a))
                → (∀ b → Q b → P (apply (α ⁻¹) b))
                → Σ A P ≃ Σ B Q
restricted-equiv α {P = P} {Q = Q} prop-P prop-Q ϵ δ
        = equiv-Σ α
           (\p → apply ((prop-≃-is-↔ (prop-P _) (prop-Q _))⁻¹)
                       (transport (\p' → P (apply (α ⁻¹) p) → Q p')
                                  (<–-inv-r α p) (ϵ _) , δ _) )
```

5.2 Propositions/Existensial.magda

This module implements the truncated existensial quantifier, using the HoTT library truncations.

```
module Propositions.Existensial where
open import lib.Base
open import lib.NType

open import lib.types.Truncation
open import lib.types.Sigma
open import lib.types.Pi
open import lib.types.TLevel
```

5.2.1 Truncated Existensial quantification

```
module _ {i j} (A : Type i) (B : A → Type j) where

  ∃ : Type (lmax i j)
  ∃ = Trunc (0 -1) (Σ A B)

  ∃-elim : ∀ {j} {P : ∃ → Type j}
         → (p : (x : ∃) → is-prop (P x)) (d : (a : Σ A B) → P [ a ])
         → Π ∃ P
  ∃-elim p = Trunc-elim p

  ∃-is-prop : is-prop ∃
  ∃-is-prop = Trunc-level
```

5.3 Propositions/Disjunction.magda

This module implements both disjoint union and truncated binary disjunction. We use the HoTT library truncations.

```
module Propositions.Disjunction where
open import lib.Base
open import lib.NType

open import lib.types.Truncation
open import lib.types.Sigma
open import lib.types.Pi
open import lib.types.TLevel
```

5.3.1 Binary disjoint union

```
module _ {i j} where

  data _+_ (A : Type i) (B : Type j) : Type (lmax i j) where
     left : A → A + B
     right : B → A + B

  cases : ∀ {k} {A : Type i} {B : Type j}
        →  {C : A + B → Type k}
        → ((a : A) → C (left a))
        → ((b : B) → C (right b))
        → (x : A + B) → C x
  cases f g (left a) = f a
  cases f g (right b) = g b
```

5.3.2 Binary truncated disjunction

```
  _∨_ : (A : Type i) (B : Type j) → Type (lmax i j)
  A ∨ B = Trunc (0 -1) (A + B)

  ∨-elim : ∀ {k} {A : Type i} {B : Type j} {P : A ∨ B → Type k}
         → (p : (x : A ∨ B) → is-prop (P x)) (d : (a : A + B) → P [ a ])
         → Π (A ∨ B) P
  ∨-elim p d = Trunc-elim p d

  ∨-elim' : ∀ {k} {A : Type i} {B : Type j} {P : A ∨ B → Type k}
         → (p : (x : A ∨ B) → is-prop (P x))
         → (d : (a : A) → P [ left a ])
         → (d' : (b : B) → P [ right b ])
         → Π (A ∨ B) P
  ∨-elim' p d d' = Trunc-elim p (cases d d')

  ∨-is-prop : (A : Type i) (B : Type j) → is-prop (A ∨ B)
  ∨-is-prop A B = Trunc-level
```

5.4 W/W.magda

```
{-# OPTIONS --without-K #-}
module W.W where

open import lib.Base
open import lib.Equivalences
```

5.4.1 W-types

This module defines W-types and characterizes their identity type.

```
data W {i j} {A : Type i} (B : A → Type j) : Type (lmax i j) where
 sup : (a : A) → (B a → W B) → W B

shape : ∀ {i j} {A : Type i} {B : A → Type j} → W B → A
shape (sup a _) = a

subtree : ∀ {i j} {A : Type i} {B : A → Type j} → (x : W B) → B (shape x) → W B
```

```
subtree (sup _ f) = f

open W
```

This is the unpacked equality on W-types:

```
W-unpacked-id : forall {i j} {A : Type i} {B : A → Type j}
              → (x y : W B) → Type (lmax i j)
W-unpacked-id {B = B} (sup a f) (sup a' f')
    = Σ (a == a')
        (\α → (f' ∘ (transport B α)) == f )
```

Home helper functions

```
id-to-W-helper : forall {i j} {A : Set i} (B : A → Set j)
               → {a a' : A} → {f : B a → W B} {f' : B a' → W B}
               → (sup a f) == (sup a' f') → W-unpacked-id (sup a f) (sup a' f')
id-to-W-helper B idp = (idp , idp)

id-to-W : forall {i j} {A : Set i} (B : A → Set j) → {x y : W B}
              → x == y → W-unpacked-id x y
id-to-W B {x = (sup a f)} {y = (sup a' f')} p = id-to-W-helper B p

W-to-id-helper : forall {i j} {A : Set i} (B : A → Set j)
               → {a a' : A} → {f : B a → W B} {f' : B a' → W B}
               → W-unpacked-id (sup a f) (sup a' f')
               → (sup a f) == (sup a' f')
W-to-id-helper B (idp , idp) = idp


W-to-id : forall {i j} {A : Set i} (B : A → Set j)
        → {x y : W B}
        → W-unpacked-id x y → x == y
W-to-id B {x = sup a f} {y = sup a' f'} α = W-to-id-helper B α
```

The main equivalence:

```
W-id-≃ : forall {i j} {A : Set i} (B : A → Set j) → {x y : W B}
              → (x == y) ≃ W-unpacked-id x y
W-id-≃ {A = A} B {sup a f} {sup a' f'} =
  ((id-to-W B) , is-eq _ (W-to-id B) (ϵ {f = f} {f' = f'}) δ) where
   ϵ : {a a' : A} {f : B a → W B} {f' : B a' → W B}
      → (b : W-unpacked-id (sup a f) (sup a' f'))
      → id-to-W B (W-to-id B {sup a f} {sup a' f'} b) == b
   ϵ (idp , idp) = idp

   δ : {a a' : A} {f : B a → W B} {f' : B a' → W B}
      → (a : (sup a f) == (sup a' f'))
      → W-to-id B (id-to-W B a) == a
   δ idp = idp
```

5.4.2 W-induction

```
W-induction : ∀ {i j k} {A : Set i} (B : A → Set j)
            → (P : W B → Set k)
            → ((x : W B) → ((i : B (shape x)) → P (subtree x i)) → P x)
            → (x : W B) → P x
W-induction B P ϕ (sup a f) = ϕ (sup a f) (\i → W-induction B P ϕ (f i))
```

5.5 Function/Fiberwise.magda

```
{-# OPTIONS --without-K #-}

module Function.Fiberwise where
```

This module gives an equivalence between "slice functions" and fibrewise functions.

```
open import lib.Base
open import lib.Equivalences
open import lib.Equivalences2
open import lib.Funext
open import lib.types.Sigma
open import lib.types.Pi
open import lib.PathGroupoid
open import lib.NType
open import Propositions.Equivalences
```

5.5.1 The fibre of a function in a point

```
fibre : ∀ {i j} {A : Type i} {B : Type j} → (f : A → B) → B → Type (lmax i j)
fibre {A = A} f b = Σ A (\a → f a == b)
```

The following lemma says how fibres behave over paths:

```
fibre-transport : ∀ {i j} {A : Type i} {B : Type j} → (f : A → B)
                → {b b' : B} → (h : b == b')
                → ∀ a e → (a , e) == (a , e • h) [ fibre f ↓ h ]
fibre-transport f idp a idp = idp
```

5.5.2 Dependent paths

```
ap-idp : ∀ {i j} {A : Type i} {B : Type j}
     → (f : A → B)
     → {a a' : A}
     → (p : a == a')
     → ap f p == idp [ (\v → f v == f a') ↓ p ]
ap-idp f idp = idp
```

The following lemma is a bit of a brute force path calculation:

```
ap-idp' : ∀ {i j} {A : Type i} {B : Type j}
     → (f r : A → B)
     → (σ : ∀ a → f a == r a)
     → {a a' : A}
     → (p : a' == a)
     → (! (σ a') • ap f p) •' (σ a) == idp [ (\v → r v == r a) ↓ p ]
ap-idp' f r σ {a = a} idp = ap (\x → x •' σ a)
                               (•-unit-r (! (σ a))) • (!-inv'-l (σ a))
```

The following lemma explains how `ap` works with compositions.

```
ap-comp : ∀ {i j k} {A : Type i} {B : Type j} {C : Type k}
        → (f : A → B) (g : B → C)
        → {a a' : A} (p : a == a')
        → ap g (ap f p) == ap (\a → g (f a)) p
ap-comp f g idp = idp

ap2d : ∀ {i j k} {A : Type i} {B : A → Type j} {C : Type k}
     → (F : ∀ a → B a → C)
     → {a a' : A} {b : B a} {b' : B a'}
     → (p : a == a') (q : b == b' [ B ↓ p ])
     → F a b == F a' b'
ap2d F idp idp = idp
```

5.5.3 Algebraic lemmas

Two general lemmas from $\prod$ and $\sum$ algebra.

```
sum-commute : ∀ {i j k} {A : Type i} (B : A → Type j) (C : A → Type k)
            → Σ (Σ A B) (C ∘ fst) ≃ Σ (Σ A C) (B ∘ fst)
sum-commute {A = A} B C = there , is-eq there back there-back back-there where
   there : Σ (Σ A B) (C ∘ fst) → Σ (Σ A C) (B ∘ fst)
   there ((a , b) , c) = ((a , c ) , b)
   back : Σ (Σ A C) (B ∘ fst) → Σ (Σ A B) (C ∘ fst)
   back  ((a , c ) , b) = ((a , b) , c)
   there-back : ∀ acb → there (back acb) == acb
   there-back ((a , c) , b) = idp
   back-there : ∀ abc → back (there abc) == abc
   back-there ((a , b) , c) = idp

prod-commute : ∀ {i j k} {A : Type i}
             → (B : A → Type j) (C : {a : A} → B a → Type k)
             → (Σ (Π A B) (\f → Π A (C ∘ f))) ≃ (Π A (\x → Σ (B x) C))
prod-commute {A = A} B C = there , is-eq there back there-back back-there where
   there : (Σ (Π A B) (\f → Π A (C ∘ f))) → (Π A (\x → Σ (B x) C))
   there (f , s) x = (f x , s x)
   back : (Π A (\x → Σ (B x) C)) → (Σ (Π A B) (\f → Π A (C ∘ f)))
   back  F = (\x → fst (F x)) , (\x → snd (F x))
   there-back : ∀ F → there (back F) == F
   there-back F = idp
   back-there : ∀ fs → back (there fs) == fs
   back-there fs = idp
```

5.5.4 Fibrewise functions

Here comes the advertised equivalence:

```
module _ {i j k} {A : Type i} {B : Type j} {C : Type k}
         (f : A → C) (g : B → C) where
  Over : Type (lmax (lmax i j) k)
  Over = (Σ (A → B) (\α → ∀ x → g (α x) == f x))

  Fibrewise : Type (lmax (lmax i j) k)
  Fibrewise = (x : C) → fibre f x → fibre g x

  over-to-fibrewise : Over → Fibrewise
  over-to-fibrewise (α , σ) c (a , p) = (α a , σ a •' p)
```

```
  fibrewise-to-over : Fibrewise → Over
  fibrewise-to-over F = (α , σ) where
     α : A → B
     α a = fst (F (f a) (a , idp))
     σ : (a : A) → g (α a) == f a
     σ a = snd (F (f a) (a , idp))

  fibrewise : Over ≃ Fibrewise
  fibrewise = over-to-fibrewise , is-eq _ fibrewise-to-over ϕ ψ where
     ϕ : ∀ F → over-to-fibrewise (fibrewise-to-over F) == F
     ϕ F = λ= (\c → (λ= (w c))) where
        w : ∀ c ap → over-to-fibrewise (fibrewise-to-over F) c ap == F c ap
        w .(f a) (a , idp) = idp
     ψ : ∀ ασ → fibrewise-to-over (over-to-fibrewise ασ) == ασ
     ψ (α , σ) = idp
```

5.5.5 Fibrewise equivalences

Now we should show that [fibrewise] preserves equivalences. The proofs here involve some ugly path computations.

```
module _ {i j k} {A : Type i} {B : Type j} {C : Type k}
         (f : A → C) (g : B → C) where
  fibrewise-equiv₀ : ∀ ασ → is-equiv (apply ασ)
                   → (∀ c → is-equiv (apply (fibrewise f g) ασ c))
  fibrewise-equiv₀ (α , σ) e c = is-eq (F c) (F' c) (F-F' c) (F'-F c) where
        α' = is-equiv.g e
        σ' : ∀ b → f (α' b) == g b
        σ' b = ! (σ (α' b)) • (ap g (is-equiv.f-g e b))
        F  = apply (fibrewise f g) (α , σ)
        F' = apply (fibrewise g f) (α' , σ')
        F-F' : ∀ c bp → F c (F' c bp) == bp
        F-F' .(g b) (b , idp) = pair= (is-equiv.f-g e b)
                                      (    use-•'=•
                                       •ᵈ (use-•-assoc
                                       •ᵈ (use-!-inv-r
                                       •ᵈ ap-idp g (is-equiv.f-g e b)))) where
            use-•'=• : (σ (is-equiv.g e b)
                       •' ! (σ (is-equiv.g e b))
                       • ap g (is-equiv.f-g e b)) ==
                              (σ (is-equiv.g e b)
                               • ! (σ (is-equiv.g e b))
                               • ap g (is-equiv.f-g e b))
            use-•'=• = •'=• (σ (is-equiv.g e b) )
                            (! (σ (is-equiv.g e b)) • ap g (is-equiv.f-g e b))
            use-•-assoc : σ (is-equiv.g e b)
                        • ! (σ (is-equiv.g e b))
                        • ap g (is-equiv.f-g e b) ==
                               (σ (is-equiv.g e b)
                                • ! (σ (is-equiv.g e b)))
                                • ap g (is-equiv.f-g e b)
            use-•-assoc = ! (•-assoc (σ (is-equiv.g e b))
                                        (! (σ (is-equiv.g e b)))
                                        (ap g (is-equiv.f-g e b)))
            use-!-inv-r : (σ (is-equiv.g e b)
                        • ! (σ (is-equiv.g e b)))
```

```
                    • ap g (is-equiv.f-g e b) == ap g (is-equiv.f-g e b)
        use-!-inv-r = ap (\x → x • ap g (is-equiv.f-g e b))
                         (!-inv-r (σ (is-equiv.g e b)))
    F'-F : ∀ c ap → F' c (F c ap) == ap
    F'-F .(f a) (a , idp) =
        pair= (is-equiv.g-f e a)
              (ap (\x → ((! (σ (is-equiv.g e (α a))) • ap g x) •' σ a))
                  (! (is-equiv.adj e a))
                   •ᵈ (ap (\x → ((((! (σ (is-equiv.g e (α a))) • x) •' σ a)))
                              (ap-comp α g (is-equiv.g-f e a))
                          •ᵈ (ap-idp' (\a → g (α a)) f σ (is-equiv.g-f e a))))

fibrewise-equiv₁ : ∀ F → (∀ c → is-equiv (F c))
                 → is-equiv (apply (fibrewise-to-over f g F))
fibrewise-equiv₁ F ϵ = is-eq α α' α-α' α'-α where
    F' = \c bp → is-equiv.g (ϵ c) bp
    α = apply (fibrewise-to-over f g F)
    α' = apply (fibrewise-to-over g f F')
    α-α' : ∀ b → α (α' b) == b
    α-α' b = ap2d (\c ap → fst (F c ap)) p (fibre-transport f p a idp)
             • (ap fst (is-equiv.f-g (ϵ c) x)) where
                 c = g b
                 x = (b , idp)
                 a = α' b
                 p : f a == c
                 p = (snd (F' c x))
    α'-α : ∀ a → α' (α a) == a
    α'-α a = ap2d (\c bp → fst (F' c bp)) p (fibre-transport g p b idp)
             • (ap fst (is-equiv.g-f (ϵ c) x)) where
                 c = f a
                 x = (a , idp)
                 b = α a
                 p : g b == c
                 p = (snd (F c x))

fibrewise-equiv : ∀ ασ → is-equiv (apply ασ)
                       ↔ (∀ c → is-equiv (apply (fibrewise f g) ασ c))
fibrewise-equiv ασ
   = fibrewise-equiv₀ ασ ,
     \ϵ → transport (\ασ → is-equiv (apply ασ))
                    (<–-inv-l (fibrewise f g) ασ)
                    (fibrewise-equiv₁ _ ϵ)
```

Now we stitch together everything to the main result:

```
≃-Over : Type (lmax (lmax i j) k)
≃-Over = (Σ (A ≃ B) (\α → ∀ x → g (apply α x) == f x))

≃-Fibrewise : Type (lmax (lmax i j) k)
≃-Fibrewise = (x : C) → fibre f x ≃ fibre g x


Over-Fibrewise-≃ : ≃-Over ≃ ≃-Fibrewise
Over-Fibrewise-≃ = (fibrewise-eq) ○e (over-fibrewise) ○e (over-equiv) where
   over-equiv : ≃-Over ≃ Σ (Over f g) (is-equiv ○ apply)
   over-equiv = (sum-commute _ _) ⁻¹
   over-fibrewise : Σ (Over f g) (is-equiv ○ apply)
                  ≃ Σ (Fibrewise f g) (\F → (∀ c → is-equiv (F c)) )
```

```
over-fibrewise = restricted-equiv (fibrewise f g)
                                  (is-equiv-is-prop ∘ apply)
                                  (Π-level ∘ (_∘_ is-equiv-is-prop))
                                  fibrewise-equiv₀ fibrewise-equiv₁
fibrewise-eq : Σ (Fibrewise f g) (\F → (∀ c → is-equiv (F c)) )
               ≃ ≃-Fibrewise
fibrewise-eq = (prod-commute _ _)
```

5.5.6 Another useful lemma

This one is from the book

```
collect-fibres : ∀ {i j} {A : Type i} {B : Type j}
                 → (f : A → B) → A ≃ (Σ B (fibre f))
collect-fibres {A = A} f = there , is-eq _ back ω (\a → idp) where
   there = \a → (f a , a , idp)
   back : (Σ _ (fibre f)) → A
   back p = fst (snd p)
   ω : ∀ bap → there (back bap) == bap
   ω (.(f a) , a , idp) = idp
```

5.6 Multiset/Iterative.magda

```
module Multiset.Iterative where

open import lib.Base
open import lib.Equivalences
open import lib.Funext
open import lib.Univalence
open import lib.types.Sigma
open import lib.types.Pi
open import lib.PathGroupoid
open import W.W
open import Function.Fiberwise
open import Propositions.Equivalences
```

We define an explicit decoding function for each Type i universe.

```
T : (i : ULevel) → Type i → Type i
T i A = A
```

5.6.1 The type of iterative multisets.

```
M : (i : ULevel) → Type (lsucc i)
M i = W (T i)

index : ∀ {i} → M i → Set i
index (sup A _) = A

element : ∀ {i} → (x : M i) → index x → M i
element (sup A f) = f
```

5.6.2 The natural equality on M

```
Eq : ∀ {i} → M i → M i  → Set i
Eq (sup A f) (sup B g) =
   Σ (A ≃ B)
     (\α → ((x : A) → Eq (g (apply α x)) (f x)))
```

5.6.3 Membership:

x ∈ y is the set of occurences of x in y

```
_∈_ : ∀ {i} → M i → M i → Set i
x ∈ (sup A f) = Σ A (\i → Eq (f i) x)

_ϵ_ : ∀ {i} → M i → M i → Set (lsucc i)
x ϵ y = Σ (index y) (\i → (element y i) == x)

Eq-refl : ∀ {i} → {x : M i} → Eq x x
Eq-refl {_} {sup _ _} = ( ide _ , \x → Eq-refl)

Id-to-Eq : (i : ULevel) {x y : W (T i)} → x == y → Eq x y
Id-to-Eq i idp = Eq-refl
```

5.6.4 A version of Eq which lives at a higher ULevel

```
Eq' : ∀ {i} → M i → M i  → Set (lsucc i)
Eq' (sup A f) (sup B g) =
   Σ (A ≃ B)
     (\α → ((x : A) → Eq' (g (apply α x)) (f x)))
```

Changing the ULevel doesn't change anything:

```
Eq'-to-Eq : {i : ULevel} (x y : M i) → Eq' x y → Eq x y
Eq'-to-Eq (sup A f) (sup B g) (α , σ) = (α , \a → Eq'-to-Eq _ _ (σ _))

Eq'-is-Eq : {i : ULevel} (x y : M i) → Eq' x y ≃ Eq x y
Eq'-is-Eq (sup A f) (sup B g)
          = equiv-Σ-snd (\α → equiv-Π-r (\x → Eq'-is-Eq _ _))
```

5.6.5 Id is Eq

We can now prove that Id is equivalent to Eq'

```
Id-is-Eq' : (i : ULevel) (x y : M i) → (x == y) ≃ Eq' x y
Id-is-Eq' i (sup A f) (sup B g) = IH ∘e EXT ∘e UA ∘e WLEM where

   IH : Σ (A ≃ B) (\α → ((x : A) → (g (apply α x)) == (f x)))
      ≃ Eq' (sup A f) (sup B g)
   IH = equiv-Σ-snd (\α → equiv-Π (ide _)
                                  (\x → Id-is-Eq' i (g (apply α x)) (f x)))

   EXT : Σ (A ≃ B) (\α → (g ∘ (apply α) == f))
       ≃ Σ (A ≃ B) (\α → ((x : A) → (g (apply α x)) == (f x)))
   EXT = equiv-Σ-snd (\α → app=-equiv)
```

```
  lem0 : {A : Type i} → (ua (ide A)) == idp
  lem0 = ua-η idp
  lem = equiv-induction
        (\α → transport (T i) (fst ua-equiv α) == apply α)
        (\A → ap (\f → coe (ap (T i) f)) lem0)
  UA : (Σ (A == B) (\α → (g ∘ (transport (T i) α) == f)))
     ≃  Σ (A ≃ B)  (\α → (g ∘ (apply α) == f))
  UA = (equiv-Σ-snd (\α → coe-equiv (ap (\h → g ∘ h == f) (lem α))))
    ∘e (equiv-Σ-fst (\α → g ∘ (transport (T i) α) == f)(snd ua-equiv)) ⁻¹
  WLEM : ((sup A f) == (sup B g))
       ≃ Σ (A == B) (\α → (g ∘ (transport (T i) α) == f))
  WLEM = W-id-≃ (T i)
```

...and thus that Id is equivalent to Eq.

```
Id≃Eq : (i : ULevel) (x y : M i) → (x == y) ≃ Eq x y
Id≃Eq = \i x y → Eq'-is-Eq x y ∘e Id-is-Eq' i x y
```

A useful consequence is that ϵ and $\in$ are equivalent

```
ϵ≃∈ : ∀ {i} {x y : M i} → (x ϵ y) ≃ (x ∈ y)
ϵ≃∈ {y = sup A f} = equiv-Σ-snd (\i → Id≃Eq _ _ _)
```

5.6.6 Extensional equality on multisets

```
ExtEq : ∀ {i} → (x y : M i) → Set (lsucc i)
ExtEq x y = ∀ z → (z ϵ x) ≃ (z ϵ y)

Extensionality : ∀ {i} → (x y : M i) → (x == y) ≃ ExtEq x y
Extensionality (sup A f) (sup B g)
    = Over-Fibrewise-≃ f g ∘e ϕ ∘e Id≃Eq _ _ _ where
      ϕ : Eq (sup A f) (sup B g) ≃ ≃-Over f g
      ϕ = equiv-Σ-snd (\α → equiv-Π (ide _) (\x → (Id≃Eq _ _ _)⁻¹))
```

5.7 Sets/Iterative.magda

This module defines and proves the basic properties of iterative sets.

```
{-# OPTIONS --without-K #-}

module Sets.Iterative where

open import lib.Base
open import lib.Equivalences
open import lib.Funext
open import lib.Univalence
open import lib.types.Sigma
open import lib.types.Pi
open import lib.PathGroupoid
open import lib.NType
open import lib.NType2
open import W.W
open import Function.Fiberwise
open import Propositions.Equivalences

import Multiset.Iterative as M
```

5.7.1 Noation

```
T = M.T

M = M.M
```

5.7.2 Definition of iterative sets

A multiset is an iterative set if elementhood is propositional and all elements are iterative sets as well.

```
is-iterative-set : ∀ {i} → M i → Type (lsucc i)
is-iterative-set (sup a f)
    = (i : T _ a) → (is-contr (Σ _ \(j : T _ a) → f j == f i)
    × is-iterative-set (f i))


iterative-set-is-prop : ∀ {i} → (x : M i) → is-prop (is-iterative-set x)
iterative-set-is-prop (sup a f)
      = Π-level (\i → ×-level is-contr-is-prop (iterative-set-is-prop (f i)))
```

5.7.3 The collection of iterative sets

```
V : ∀ i → Type (lsucc i)
V i = Σ (M i) is-iterative-set
```

5.7.4 Accessing the various parts of a multiset.

```
underlying-M : ∀ {i} → V i → M i
underlying-M = fst


index : ∀ {i} → V i → Set i
index x = M.index (underlying-M x)

element : ∀ {i} → (x : V i) → index x → V i
element (sup a f , e) i = (f i , snd (e i))
```

The underlying multiset of an element is the elment of the underlying multiset:

```
element-underlies : ∀ {i} → (x : V i) → (i : index x)
                  → M.element (underlying-M x) i == underlying-M (element x i)
element-underlies (sup a f , e) i = idp
```

5.7.5 Two equivalent forms of elementhood

```
_∈_ : ∀ {i} → V i → V i → Type i
x ∈ y = underlying-M x M.∈ underlying-M y

_ϵ_ : ∀ {i} → V i → V i → Type (lsucc i)
x ϵ y = underlying-M x M.ϵ underlying-M y
```

Given an element of the index of a set, we get an actual element.

```
elements-ϵ : ∀ {i} → (x : V i) → (i : index x) → (element x i ϵ x)
elements-ϵ (sup A f , p) i = (i , idp)

itset-element-is-itset : ∀ {i} → (x : V i) → (z : M i)
                       → z M.ϵ (underlying-M  x) → is-iterative-set z
itset-element-is-itset (sup A f , i) z (a , p)
         = transport is-iterative-set p (snd (i a))

underlying-full-and-faitful : ∀ {i} → (x y : V i)
                            → (underlying-M x == underlying-M y) ≃ (x == y)
underlying-full-and-faitful (x , p) (y , p')
         = prop-equal-≃ iterative-set-is-prop p p'
```

5.7.6 Elementhood in itsets is propositional

```
ϵ-is-prop : ∀ {i} → (x : M.M i) (y : V i) → is-prop (x M.ϵ underlying-M y)
ϵ-is-prop x (sup a f , p)
   = inhab-to-contr-is-prop (\e → transport (\x → is-contr (x M.ϵ (sup a f)))
                                            (snd e)
                                            (fst (p (fst e))))
```

5.7.7 Extensionality for sets

```
ExtEq : ∀ {i} → V i → V i → Type (lsucc i)
ExtEq x y = ∀ z → z ϵ x ↔ z ϵ y

ExtEqM : ∀ {i} → V i → V i → Type (lsucc i)
ExtEqM x y = ∀ z → z M.ϵ (underlying-M x) ↔ z M.ϵ (underlying-M y)

ExtEq-is-prop : ∀ {i} (x y : V i) → is-prop (ExtEq x y)
ExtEq-is-prop x y = Π-level (\z → ↔-level (ϵ-is-prop (underlying-M z) x)
                                          (ϵ-is-prop (underlying-M z) y))

ExtEqM-is-prop : ∀ {i} (x y : V i) → is-prop (ExtEqM x y)
ExtEqM-is-prop x y = Π-level (\z → ↔-level (ϵ-is-prop z x)
                                           (ϵ-is-prop z y))

ExtEqM≃ExtEq : ∀ {i} → (x y : V i) → ExtEqM x y ≃ ExtEq x y
ExtEqM≃ExtEq x y = apply (prop-≃-is-↔ (ExtEqM-is-prop x y)
                                      (ExtEq-is-prop x y) ⁻¹)
                         (ϕ , ψ) where
     ϕ : ExtEqM x y → ExtEq x y
     ϕ p = p ∘ underlying-M
     ψ : ExtEq x y → ExtEqM x y
     ψ f z = (\q → (fst (f (z , itset-element-is-itset x z q)) q))
           , (\q → snd (f (z , itset-element-is-itset y z q)) q)

Extensionality : ∀ {i} (x y : V i) → (x == y) ≃ ExtEq x y
Extensionality x y = (ExtEqM≃ExtEq x y)
                   ∘e ϕ
                   ∘e M.Extensionality _ _
                   ∘e underlying-full-and-faitful _ _ ⁻¹ where
    ϕ : M.ExtEq (underlying-M x) (underlying-M y) ≃ ExtEqM x y
    ϕ = (equiv-Π (ide _) (\z → prop-≃-is-↔ (ϵ-is-prop z x) (ϵ-is-prop z y) ))
```

Equality of itsets is propositional, thus the collection of all iterative sets is a set:

```
==-is-prop : ∀ {i} (x y : V i) → is-prop (x == y)
==-is-prop x y = equiv-preserves-level (Extensionality x y ⁻¹)
                                       (ExtEq-is-prop x y)

V-is-set : ∀ {i} → is-set (V i)
V-is-set = ==-is-prop
```

We need equality to be small, so we define a small version

```
_=e_ : ∀ {i} → V i → V i → Type i
x =e y = M.Eq (underlying-M x) (underlying-M y)

=e-to-== : ∀ {i} (x y : V i) → (x =e y) ≃ (x == y)
=e-to-== x y = underlying-full-and-faitful x y ∘e (M.Id≃Eq _ _ _ ⁻¹)
```

5.7.8 Constructing sets by small injections

```
_#_ : ∀ {i} {A : Type i}
  → (f : A → V i) → (∀ a a' → (a == a') ≃ (f a == f a'))
  → V i
f # e = (sup _ (underlying-M ∘ f)
      , \a → (equiv-preserves-level
                  (equiv-Σ-snd (\a' → (underlying-full-and-faitful _ _ ⁻¹)
                                    ∘e e a' a))
                  (pathto-is-contr a) , snd (f a) ))


_-_#_ : ∀ {i} {A : Type i}
  → (is-set A)
  → (f : A → V i) → (∀ a a' → (f a == f a') → a == a')
  → V i
p - f # e = f # (\a a' → apply (prop-≃-is-↔ (p _ _)
                                            (V-is-set _ _) ⁻¹)
                                            (ap f , e a a'))
```

5.7.9 The index of an iterative set is a set

```
index-is-set : ∀ {i} → (x : V i) → is-set (index x)
index-is-set (sup a f , p)
     = equiv-preserves-level (collect-fibres (element (sup a f , p)) ⁻¹
                                ∘e (equiv-Σ-snd
                                     (\y → equiv-Σ-snd
                                       (\i → underlying-full-and-faitful _ _))))
                             (Σ-level
                                   V-is-set
                                   (\y → prop-is-set (ϵ-is-prop (underlying-M y)
                                                                (sup a f , p))))
```

5.7.10 Eliminating set quotients to propositions

These lemma are general, but are put here in lack of a better place.

```
open import lib.types.SetQuotient

SetQuot-prop-elim : ∀ {i j k} {A : Type i} {R : A → A → Type j}
                  → (P : SetQuotient R → Type k)
                  → Π _ (is-prop ∘ P) → (∀ a → P q[ a ])
                  → Π _ P
SetQuot-prop-elim P p q
    = SetQuot-elim (prop-is-set ∘ p)
                   q
                   (\r → fst (is-prop-has-contr-path-over p _ _ _)) where

SetQuot-prop-elim2 : ∀ {i j k} {A : Type i} {R : A → A → Type j}
                  → (P : SetQuotient R → SetQuotient R → Type k)
                  → (∀ a b → is-prop (P a b)) → (∀ a b → P q[ a ] q[ b ])
                  → (∀ a b → P a b)
SetQuot-prop-elim2 P p q = SetQuot-prop-elim
           (\a → ∀ b → P a b) (\a → (Π-level (\b → p a b)))
           \a → SetQuot-prop-elim (\ b → P q[ a ] b) (\b → p q[ a ] b )
             \b → q a b
```

5.7.11 Quotient lemma

This lemma goes into constructing pairs and unions.

```
module _ {i} {A : Set i} (f : A → V i) where
  self-equaliser : Set i
  self-equaliser = SetQuotient (\a b → f a == f b)

  inject : self-equaliser → V i
  inject = SetQuot-rec (V-is-set {i}) f (\e → e)

  inject-inj : ∀ a b → inject a == inject b → a == b
  inject-inj = SetQuot-prop-elim2 (\a b → inject a == inject b → a == b)
                                  (\a b → Π-level (\e → SetQuotient-is-set a b))
                                   \a b → quot-rel

  quotient-set : V i
  quotient-set = SetQuotient-is-set - inject # inject-inj


  quotient-bound : ∀ {j} {P : V i → Type j}
                 → (Π _ (is-prop ∘ P))
                 → ((a : A) → P (f a))
                 → ∀ z → (z ϵ quotient-set) → P z
  quotient-bound {P = P} p r z (qa , e)
          = transport P (apply (underlying-full-and-faitful _ _) e)
                        (SetQuot-prop-elim _ (\c → p (inject c)) r qa )
```

5.7.12 Iterated quotients

```
iterated-quotient : ∀ {i} → M i → V i
iterated-quotient (sup a f) = quotient-set (\i → iterated-quotient (f i))
```

5.7.13 Truncated Aczel equality

The following relation gives an alternative way of constructing V as a quotient of M rather than as a subtype.

```
open import Propositions.Equivalences
open import Propositions.Disjunction
open import Propositions.Existensial
open import lib.types.Truncation

truncated-aczel-equality : ∀ {i} (x y : M i) → Type i
truncated-aczel-equality (sup a f) (sup b g)
   = (∀ x → ∃ _ (\y → truncated-aczel-equality (f x) (g y)))
   × (∀ y → ∃ _ (\x → truncated-aczel-equality (f x) (g y)))

_≈_ = \{i} (x y : M i) → truncated-aczel-equality x y


≈-to-== : ∀ {i} (x y : M i) → (p : is-iterative-set x) (q : is-iterative-set y)
          → x ≈ y → x == y
≈-to-== {i} (sup a f) (sup b g) p q r
     = apply (((Extensionality x y) ∘e (underlying-full-and-faitful _ _)) ⁻¹)
        (\z → (\{(i , α) →
                       ∃-elim _ _ (\_ → ϵ-is-prop _ y)
                              ((transport (\w → w M.ϵ (sup b g) ) α)
                               ∘ (ih₁ (f i) (snd (p i))))
                              (fst r i) }) ,
              \{(j , β) → ∃-elim _ _ (\_ → ϵ-is-prop _ x)
                              ((transport (\w → w M.ϵ (sup a f) ) β)
                               ∘ (ih₀ (g j) (snd (q j))))
                              (snd r j) }) where
   ih₁ : ∀ w → (iw : is-iterative-set w)
       → Σ b (\j → w ≈ g j) → Σ b (\j → g j == w )
   ih₁ w iw (j , e) = (j , ! (≈-to-== w (g j) iw (snd (q j)) e))
   ih₀ : ∀ w → (iw : is-iterative-set w)
       → Σ a (\i → f i ≈ w) → Σ a (\i → f i == w)
   ih₀ w iw (i , e) = (i , ≈-to-== (f i) w (snd (p i)) iw e)
   x : V i
   x = sup a f , p
   y : V i
   y = sup b g , q


≈-to-iterated-quotient : ∀ {i} (x : M i)
                       → x ≈ (underlying-M (iterated-quotient x))
≈-to-iterated-quotient (sup a f)
         = (\i → [ (q[ i ] , ≈-to-iterated-quotient (f i))  ])
         , (SetQuot-prop-elim (\j → _)
                              (\_ → ∃-is-prop _ _)
                              (\j → [( j , ≈-to-iterated-quotient (f j))]))
```

5.8 Sets/Axioms.magda

This module proves various axioms of constructive set theory for our model.

```
{-# OPTIONS --without-K #-}


module Sets.Axioms where

open import Sets.Iterative
open import lib.Base
open import lib.Equivalences
open import lib.NType
open import lib.PathGroupoid
open import lib.Funext

open import lib.types.Span
open import lib.types.SetQuotient
open import lib.types.Pi
open import lib.types.Sigma
open import lib.types.Truncation

open import Propositions.Equivalences
open import Propositions.Disjunction
open import Propositions.Existensial

open import Function.Fiberwise

import Multiset.Iterative as M

open import W.W

open W
```

5.8.1 Misc

```
data Nil (i : _) : Type i where

data One (i : _) : Type i where
  ∗ : One i

One-is-contr : ∀ {i} → is-contr (One i)
One-is-contr = ∗ , \{∗ → idp}

One+One-is-set : ∀ {i} → is-set (One i + One i)
One+One-is-set (left ∗) (right ∗) ()
One+One-is-set (right ∗) (left ∗) ()
One+One-is-set (left ∗) (left ∗) idp idp = idp , \{idp → idp}
One+One-is-set (right ∗) (right ∗) idp idp = idp , \{idp → idp}


⊙ : ∀ {i j} (A : Set j) → A → One i
⊙ A _ = ∗

Nil-elim : ∀ {i j} {A : Nil i → Type j} → Π _ A
Nil-elim ()
```

5.8.2 Constructions underlying the axioms

```
∅ : ∀ {i} → V i
∅ = (sup (Nil _) Nil-elim , Nil-elim)
```

```
singleton : ∀ {i} → V i → V i
singleton x = (contr-is-set One-is-contr) - (\_ → x) # \{* * p → idp}

p : ∀ {i} → V i → V i → V i
p {i} x y = quotient-set φ where
   φ : One i + One i → V i
   φ (left _) = x
   φ (right _) = y

[z∈_‖_] : ∀{i} → V i → (Σ (V i → Type i) (\P → Π _ (is-prop ∘ P))) → V i
[z∈ (sup a f , e) ‖ ( P , p) ]
    = sup (Σ _ (P ∘ element (sup a f , e))) (f ∘ fst) ,
      (\x → inhab-prop-is-contr (x , idp)
                                (equiv-preserves-level
                                     (sum-commute _ _)
                                     (Σ-level
                                         (contr-is-prop (fst (e (fst x))))
                                         (\i → p _))) , snd (e (fst x)) ) )

⋃ : ∀ {i} → V i → V i
⋃ {i} x = quotient-set φ where
  φ : Σ (index x) (index ∘ element x) → V i
  φ (i , j) = element (element x i) j

G : ∀ {i} → (x y : V i) → (index x → index y) → V i
G x y f = quotient-set (element y ∘ f) where

γ : ∀ {i} → (x y : V i) → V i
γ x y = quotient-set (G x y)
```

Bounded Quantification C:1. Beware! Has been known to cause troubles.

```
bonded : ∀ {i j} {P : V i → Type j}
        → (x : V i)
        → ((i : index x) → P (element x i))
        → ∀ z → (z ϵ x) → P z
bonded {P = P} (sup A f , _) p z (i , e)
   = transport P (apply (underlying-full-and-faitful _ _) e) (p i)
```

5.8.3 Proof of axioms

```
PAIR : ∀ {i} → ∀ (x : V i) y → Σ _ (\u → ∀ z → z ϵ u ↔ (z == x) ∨ (z == y))
PAIR x y = ( p x y , \z → (φ z , ψ z)) where
   φ = quotient-bound _ (\_ → ∨-is-prop _ _)
                      \{(left _) → [ left idp ] ;
                        (right _) → [ right idp ] }
   ψ : ∀ z → (z == x) ∨ (z == y) → z ϵ p x y
   ψ z = ∨-elim' (\_ → (ϵ-is-prop (underlying-M z) (p x y)))
                 (\p → (q[ left *  ] , ! (ap underlying-M p)))
                 (\p → (q[ right * ] , ! (ap underlying-M p)))

RSEP : ∀ {i} → {P : V i → Type i} → (Π _ (is-prop ∘ P))
    → ∀ x → Σ _ \u → ∀ z → (z ϵ u ↔ P z × (z ϵ x))
RSEP {P = P} p (sup A f , e)
```

```
     = [z∈ (sup A f , e) ‖ (_ , p)] , \z → (ϕ z , ψ z) where
        ϕ : ∀ z
          → z ϵ [z∈ (sup A f , e) ‖ (P , p)]
          → P z × (z ϵ (sup A f , e))
        ϕ z ((a , e) , r)
            = (transport P
                         (apply (underlying-full-and-faitful _ _) r)
                         e
                , (a , r))
        ψ : ∀ z
           → P z × (z ϵ (sup A f , e))
           → z ϵ [z∈ (sup A f , e) ‖ (P , p)]
        ψ z (e , (a , r))
            = ((a , transport P
                              (apply (underlying-full-and-faitful _ _) (! r))
                              e)
                  , r)


trsp2 : ∀ {i j k} {A : Type i} {B : A → Type j} (P : (a : A) → B a → Type k)
      → {a a' : A}
      → (p : a == a')
      → (b : B a)
      → P a b → P a' (transport B p b)
trsp2 P idp b p = p

UNION : ∀ {i} → ∀ (x : V i)
      → Σ _ \y → ∀ z → (z ϵ y ↔ ∃ _ \u → (z ϵ u) × (u ϵ x))
UNION (sup A f , p) = (⋃ (sup A f , p) , \z → ( ϕ z , ψ z)) where
   ϕ = quotient-bound
         _
         (\qij → ∃-is-prop _ _)
         \{(i , j) → [ element (sup A f , p) i
                     , ((j , element-underlies _ _ ) , (i , idp)) ] }
   ψ : ∀ z → ∃ _ (λ u → (z ϵ u) × (u ϵ sup A f , p)) → z ϵ ⋃ (sup A f , p)
   ψ z = ∃-elim _ _
               (\x → ϵ-is-prop (underlying-M z) (⋃ (sup A f , p)))
               \{(u  , (j , uj=z) , (i , fi=u))
                    → (q[ i , transport M.index (! fi=u) j ]
                       , (! (element-underlies _ _))
                          • trsp2 (\x i → M.element x i == underlying-M z)
                                  (! fi=u) j uj=z) }

SCOLL : ∀ {i j} → (P : V i → V i → Type j)
      → ∀ a → (∀ x → (x ϵ a) → Σ _ (\y → P x y))
      → Σ _ (\b → (∀ x → (x ϵ a) → Σ _ (\y → (y ϵ b) × (P x y )))
                × (∀ y → (y ϵ b) → ∃ _ (\x → (x ϵ a) × (P x y ))))
SCOLL P (sup A f , p) r = (b , ϕ , ψ) where
   a = (sup A f , p)
   b = quotient-set (\i → fst (r (element a i) (i , idp)))
   ϕ : ∀ x → (x ϵ a) → Σ _ (\y → (y ϵ b) × (P x y ))
   ϕ x (i , p) = ( y
                 , (q[ i ] , idp)
                 , transport (\x → P x y)
                             (apply (underlying-full-and-faitful _ _) p)
                             (snd (r (element a i) (i , idp))) ) where
            y = fst (r (element a i) (i , idp))
   ψ : ∀ y → (y ϵ b) → ∃ _ (\x → (x ϵ a) × (P x y ))
```

```
   ψ = quotient-bound _ (\qi → ∃-is-prop _ _)
                        (\i → [ (element a i)
                              , (i , idp)
                               , snd (r (element a i) (i , idp))])

SUBC : ∀ {i j} → (P : V i → V i → V i → Type j)
     → ∀ {a b}
     → Σ _ (\c → ∀ u → (∀ x → (x ϵ a) → Σ _ (\y → (y ϵ b) × P u x y))
                     → (Σ _ (\d → (d ϵ c)
                                  × (∀ x → (x ϵ a)
                                         → ∃ _ (\y → (y ϵ d) × (P u x y )))
                                  × (∀ y → (y ϵ d)
                                         → ∃ _ (\x → (x ϵ a) × (P u x y ))))))
SUBC P {sup A f , p} {sup B g , p'} = γ a b
                                    , \u F → (d u F
                                          , (di u F , idp)
                                          , ϕ u F , ψ u F) where
  a = (sup A f , p)
  b = (sup B g , p')
  d = \u F → G a b (\i → fst (fst (snd (F (element a i) (elements-ϵ a i)))))
  di = \u F → q[ (\i → fst (fst (snd (F (element a i) (elements-ϵ a i))))) ]
  ϕ : ∀ u (F : ∀ x
             → (x ϵ a)
             → Σ _ (\y → (y ϵ b) × P u x y)) x
    → (x ϵ a)
    → ∃ _ (\y → (y ϵ d u F) × (P u x y ))
  ϕ u F (.(f i) , e) (i , idp)
    = [ (fst (F (element a i) (elements-ϵ a i)))
      , (q[ i ]
        , (snd (fst (snd (F (element a i) (elements-ϵ a i))))))
      , transport (\x → P u x (fst (F (element a i) (elements-ϵ a i))))
                  (apply (underlying-full-and-faitful _ _) idp)
                  (snd (snd (F (element a i) (elements-ϵ a i))))]
  ψ : ∀ u F y → (y ϵ d u F) → ∃ _ (\x → (x ϵ a) × (P u x y ))
  ψ u F = quotient-bound

                  _
                  (\qi → ∃-is-prop _ _)
                  (\i → [ element a i
                        , (elements-ϵ a i)
                        , transport
                             (P u (element a i))
                             (! (apply
                                    (underlying-full-and-faitful _ _)
                                    (snd (fst (snd (F (element a i)
                                                      (elements-ϵ a i)))))))
                             (snd (snd (F (element a i) (elements-ϵ a i)))) ])


ϵ-IND : ∀ {i j} (P : V i → Type j)
      → (∀ x → (∀ y → (y ϵ x) → P y) → P x)
      → (∀ x → P x)
ϵ-IND P ϕ (sup A f , e)
     = ϕ x (bonded x (\i → ϵ-IND P ϕ (f i , snd (e i)))) where
        x = sup A f , e
```

References

Cockx, Jesper, Dominique Devriese, and Frank Piessens (2014). "Pattern Matching Without K". In: *SIGPLAN Not.* 49.9, pp. 257–268. ISSN: 0362-1340. DOI: 10.1145/2692915.2628139. URL: http://doi.acm.org/10.1145/2692915.2628139.

D

Type Theoretical Databases

Henrik Forssell, Håkon Robbestad Gylterud, David I. Spivak

Abstract

We show how the display-map category of finite (symmetric) simplicial complexes can be seen as representing the totality of database schemas and instances in a single mathematical structure. We give a sound interpretation of a certain dependent type theory in this model, and show how it allows for the syntactic specification of schemas and instances and the manipulation of the same with the usual type-theoretic operations.

1 Introduction

Databases being, essentially, collections of (possibly interrelated) tables of data, a foundational question is how to best represent such collections of tables mathematically in order to study their properties and ways of manipulating them. The relational model, essentially treating tables as structures of first-order relational signatures, is a simple and powerful representation. Nevertheless, areas exist in which the relational model is less adequate than in others. One familiar example is the question of how to represent partially filled out rows or missing information. Another, more fundamental perhaps, is how to relate instances of different schemas, as opposed to the relatively well understood relations between instances of the same schema. Adding to this, an increasing need to improve the ability to relate and map data structured in different ways suggests looking for alternative and supplemental ways of modelling tables, more suitable to "dynamic" settings. It seems natural, in that case, to try to model tables of different shapes as living in a single mathematical structure, facilitating their manipulation across different schemas.

We investigate, here, a novel way of representing data structured in systems of tables which is based on simplicial sets and type theory rather than sets of relations and first-order logic[32]. Formally, we present a soundness theorem (Theorem D:23) for a certain dependent type theory with respect to a rather simple category of (finite, abstract)

[32]We thank an anonymous referee for pointing out that using the categorical semantics of type theory to structure data was an explicit motivation even at its very conception (see Cartmell 1986a,b).

simplicial complexes. An interesting type-theoretic feature of this is that the type theory has context constants, mirroring that our choice of "display maps" does not include all maps to the terminal object. From the database perspective, however, the interesting aspect is that this category can in a natural way be seen as a category of tables; collecting in a single mathematical structure—an indexed or fibered category—the totality of schemas and instances. It is the database perspective that motivates, or forces, our choice of display maps.

The representation can be introduced as follows. Let a schema S be presented as a finite set $\mathbf{A}$ of attributes and a set of relation variables over those attributes. One way of allowing for partially filled out rows is to assume that whenever the schema has a relation variable R, say over attributes $A_0, \ldots, A_n$, it also has relation variables over all non-empty subsets of $\{A_0, \ldots, A_n\}$. So a partially filled out row over R is a full row over such a "partial" relation, or "part-relation", of R. To this we add the requirement that the schema does not have two relation variables over exactly the same attributes. This requirement means that a relation variable can be identified with the set of its attributes. Together with the first condition, this means that the schema can be seen as a downward closed sub-poset of the positive power set of the set of attributes $\mathbf{A}$. Thus a schema is an (abstract) *simplicial complex*—a combinatorial and geometric object familiar from algebraic topology.

The key observation is now that an instance of the schema S can also be regarded as a simplicial complex, by regarding the data as attributes and the tuples as relation variables. Accordingly, an instance over S is a schema of its own, and the fact that it is an instance of S is "displayed" by a certain projection to S. Thus the category $\mathbb{S}$ of finite simplicial complexes and morphisms between them form a category of schemas which includes, at the same time, all instances of those schemas; where the connection between schema and instance is given by a collection D of maps in $\mathbb{S}$ called *display* maps.

We show, essentially, that $\mathbb{S}$ together with this collection D of maps form a so-called *display-map category*[33], a notion originally developed in connection with categorical models of dependent type theory. First, this means that the category $\mathbb{S}$ has a rich variety of ready-made operations that can be applied to schemas and instances. For example, the so-called dependent product operation can be seen as a generalization of the natural join operation. Second, it is a model of dependent type theory. We specify a dependent type theory and a sound interpretation which inter-

[33] Jacobs 1999.

prets contexts as schemas and types as instances. This interpretation is with respect to the display-map category $(\mathbb{S}, D)$ in its equivalent form as an indexed category. We introduce context constants interpreted as distinguished single relation variable schemas (or *relation schemas* in the terminology of Abiteboul, Hull, and Vianu 1995), reflecting the special status of such schemas.

The type theory allows for the syntactic specification of both schemas and instances. The elements of type-theoretically defined operations on these, such as the natural join, can be formally derived in the type theory. Accordingly, we see this work as being among first steps towards establishing closer links between databases and programming languages—here in the form of dependent type theories. Towards this end, we put some emphasis on showing that the model can be equipped with a type-theoretic *universe*. An infinite instance coding all finite instances of a schema, the universe allows reasoning generically about classes of instances of in the type theory itself, without having to resort to the metalanguage. Thus it provides the basis for precise, formal definitions and analyses of further database-theoretic notions (such as *query*).

The representation of tables of data presented here takes the view that tables are collections of tuples, as in the relational model. Here, this is tightly linked (in the sense of Lemma D:13) with the requirement that a schema does not have more than one relation variable over a given set of attributes[34]. An alternate view that tables are collections of "keys", with the possibility that two keys can represent the same data, and that schemas can have any number of relation variables with the same attributes, can be pursued by using (semi-)simplicial sets rather than simplicial complexes (see Spivak 2009). We take the former view in this paper as it gives a rather clear and simple picture of the representation of collections of tuples "simplicially". Also, we do not discuss extra levels of structure, like data types (as is done in ibid.), but focus on the representation of schemas and instances as simplicial complexes and their type-theoretic aspects. Section 2 introduces the category of simplicial complexes and display maps, and the representation of tables in that setting. We strive for the presentation to be as self-contained as possible, and assume for the most part only knowledge of the very basic notions of category theory, such as category, functor, and natural transformation. Section 2.2 contains the essential constructions and lemmas needed for the proof of the main soundness theorem (Theorem D:23). That theorem and the presentation of the type theory is given in Section

[34]We do not believe that this restriction is of major practical significance, see Example D:46

3. Section 4 presents the Universe, type-theoretically and semantically, and gives some illustration of its use. Finally, Section 5 contains some additional examples, informally presented. These examples are intended to supplement Section 2, to give a feel for the "simplicial" representation of tables, and to indicate uses of this representation to model such things as updates or missing data. This section can be read in parallel with Section 2.

2 The model

2.1 Complexes, schemas, and instances

We fix the following terminology and notation, adjusting the standard terminology somewhat for our purposes. A background on simplicial complexes and simplicial sets can be found in e.g. Friedman 2012; Gabriel and Zisman 1967. For symmetric simplicial sets see Grandis 2001.

A simplicial complex can be thought of as a set of vertices together with a collection of faces, where the set of faces is a downward closed set of finite non-empty, non-singleton subsets of vertices. More formally, we use the following definitions.

Based poset. Let X be a poset. A subset $B \subseteq X$ is called a *basis of* X if the following hold:

1. for all $x, y \in X$, one has $x \leq y$ if and only if $B_{\leq x} \subseteq B_{\leq y}$, where $B_{\leq x} = (\downarrow x) \cap B = \{z \in B \mid z \leq x\}$;
2. for all $g, h \in B$ one has $g \leq h \Rightarrow g = h$; and
3. $B_{\leq x}$ is inhabited and of bounded finite size for all $x \in X$. That is, there exists an $n \in \mathbb{N}$ such that for all $x \in X$, $1 \leq |B_{\leq x}| \leq n$.

If X has a basis, one sees easily that the basis is unique, and we say that X is a *based poset*.

Dimension. Let X be a based poset with basis B. Then define $X_n := \{x \in X \mid |B_{\leq x}| = n + 1\}$ to be the set of faces *of dimension* n. In particular, the set of vertices $X_0 = B$ can be considered as faces of dimension 0.

Simplicial complex. A based poset X is called a *simplicial complex* if for all $x \in X$ and $Y \subseteq B_{\leq x}$ there exists $y \in X$ such that $B_{\leq y} = Y$.

Example D:1. For a set S, the poset $\mathcal{P}_+^{\leq n}(S)$ of non-empty subsets of S of cardinality at most n is a simplicial complex. So is any downward

closed subposet of $\mathcal{P}_+^{\leq n}(S)$. If X is a simplicial complex it is isomorphic to one of this (latter) form by the map $x \mapsto B_{\leq x}$.

Simplicial schema. We say that a poset X is a *simplicial schema* if X^{op}—the poset obtained by reversing the ordering—is a simplicial complex. The elements of X_0 are called *attributes* and the elements of X_{n+1} are called *relation variables*. We consider a simplicial schema as a category and use arrows $\delta_y^x : x \longrightarrow y$ to indicate order. Thus the arrow δ_y^x exists iff $y \leq x$ in the simplicial complex X^{op}. We reserve the use of arrows to indicate order in the schema X and $\leq$ to indicate the order in the complex X^{op}. We use the notation $B_{\leq x}$ also in connection with schemas, where it means, accordingly, the set of attributes A such that there is an arrow δ_A^x.

Morphisms. Suppose that X and Y are based posets with bases B and C respectively. A poset morphism $f : X \longrightarrow Y$ is called *based* if for all $x \in X$, we have $f(B_{\leq x}) = C_{\leq f(x)}$. A morphism of simplicial complexes is a based poset morphism. A morphism of simplicial schemas is a morphism of posets $f : X \longrightarrow Y$ such that $f^{\mathrm{op}} : X^{\mathrm{op}} \longrightarrow Y^{\mathrm{op}}$ is a morphism of simplicial complexes. Note that a based poset morphism $f : X \longrightarrow Y$ is completely determined by its restriction to the basis $f_0 : X_0 \longrightarrow Y_0$.

Display maps. A morphism $f : X \longrightarrow Y$ of simplicial schemas is a *display map* if f restricts to a family of maps $f_n : X_n \to Y_n$ (one could say that it 'preserves dimension'). It is straightforward to see that this is equivalent to the condition that for all $x \in X^{\mathrm{op}}$ the restriction $f \restriction_{(\downarrow x)} : (\downarrow x) \to (\downarrow f(x))$ is an isomorphism of sets (equivalently, of simplicial complexes).

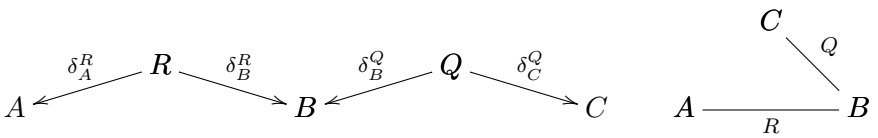
Remark D:3. With respect to the usual notion of schema, a simplicial schema X can be thought of as given in the usual way by a finite set of attributes $X_0 = \{A_0, \ldots, A_{n-1}\}$ and a set of relational variables $X = \{R_0, \ldots R_{m-1}\}$, each with a specification of column names in the form of a subset of X_0, but with the restrictions 1) that no two relation variables are over exactly the same attributes; and 2) for any nonempty subset of the attributes of a relation variable there exists a relation variable over (exactly) those attributes.

Large and small, the categories $\mathbb{S}$ and $\mathbb{S}_d$. Since we aim to represent database schemas and instances we are interested primarily in the finite case. However, we shall need to consider the infinite case in connection with type theoretical universes. Say that a simplicial schema or complex is *small* if it is finite, and *large* otherwise. For simplicity of presentation, we restrict to the finite case in the rest of this section. The restriction

is, however, mostly inessential, the definitions and lemmas generalise to the infinite case. Let $\mathbb{S}$ be the category of small simplicial schemas and morphisms and $\mathbb{S}_d$ the subcategory of small schemas and display maps. In the sequel we shall drop the word "simplicial" and simply say "complex" and "schema".

Simplices. The category $\mathbb{S}_d$ contains in particular (the opposites of) the n-simplices Δ_n and the face maps. Recall that the n-simplex Δ_n is the complex given by the full positive power set on $[n] = \{0, \ldots, n\}$. A face map $d_i^n : \Delta_n \longrightarrow \Delta_{n+1}$ between two simplices is the based poset morphism defined by $k \mapsto k$, if $k < i$ and $k \mapsto k+1$ else. These satisfy the simplicial identity $d_i^{n+1} \circ d_j^n = d_{j-1}^{n+1} \circ d_i^n$ if $i < j$. As a schema, Δ_n is the schema of a single relation on $n+1$ attributes named by numbers $0, \ldots, n$ (and all its "generated" sub-relations). A face map $d_i^n : \Delta_n \longrightarrow \Delta_{n+1}$ is the inclusion of the relation $[n+1]-\{i\}$ into $[n+1]$. These schemas and morphisms play a special role in Section 3 where they are used to specify general schemas and instances. The permutations of Δ_n are also in $\mathbb{S}_d$; we have not assumed that attributes are ordered and that (display) maps preserve order.

Example D:4. Let S be the schema with attributes A, B, C and relation variables $R : AB$ and $Q : BC$. From a "simplicial" point of view, S is the category given below left. It can also be regarded, more geometrically, as the "horn" below right.



For another example, the 2-simplex Δ_2 can be seen as a schema on attributes 0,1, and 2, with relation variables $\{0,1,2\}$, $\{0,1\}$, $\{0,2\}$, and $\{1,2\}$. The function f_0 given by $A \mapsto 0$, $B \mapsto 1$, and $C \mapsto 2$ defines a morphism $f : S \longrightarrow \Delta_2$ of schemas/complexes. f is a display map. For an example of a display map that is not an inclusion, consider the morphism $f' : S \longrightarrow \Delta_1$ defined by $A \mapsto 0$, $B \mapsto 1$, and $C \mapsto 0$

Relational instances, the categories $\mathrm{Rel}(X)$. Let X be a schema. A functor $F : X \longrightarrow \mathbf{FinSet}$ from X to the category of finite sets and functions can be regarded as an instance of the schema X. The set $F(x)$ can be regarded as a set of "keys" or "row-names"; for $A \in B_{\leq x}$ the "value" $k[A]$ of such a key $k \in F(x)$ at attribute A is the element $k[A] := F(d_A^x)(k) \in F(A)$. Accordingly, there is a mapping

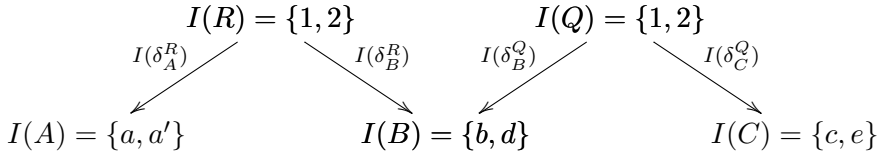
$F(x) \twoheadrightarrow \prod_{A \in B_{\leq x}} F(A)$ defined by mapping k to the function $A \mapsto k[A]$. For arbitrary F, this mapping is not 1–1, that is, there can be distinct keys with the same values at all attributes. We say that F is a *relational instance* if this does not happen. That is to say, a relational instance is a functor $F : X \longrightarrow \mathbf{FinSet}$ such that for all $x \in X$ the functions $\{F(\delta_A^x) \mid A \in B_{\leq x}\}$ are jointly injective. Let $\mathrm{Rel}(X)$ be the category of relational instances and natural transformations between them. (Notice that a natural transformation between relational instances is the same thing as a homomorphism.)

Example D:6. Let S be the schema of Example D:4. Let an instance I be given by

R	A	B
1	a	b
2	a'	b

Q	B	C
1	b	c
2	d	e

Then I is the functor



with $I(\delta_A^R)(1) = a$, $I(\delta_B^R)(1) = b$ and so on.

Example D:8. Let J be the instance $J : \Delta_2 \longrightarrow \mathbf{FinSet}$ given by $J(\{0,1,2\}) = \{\langle a,b,c\rangle\}$, $J(\{0,1\}) = \{\langle a,b\rangle\,,\ \langle a',b\rangle\}$, $J(\{1,2\}) = \{\langle b,c\rangle\,,\ \langle d,e\rangle\}$, $J(\{0,2\}) = \{\langle a,c\rangle\,,\langle a',c\rangle\}$, $J(0) = \{a, a'\,,$ $J(1) = \{b,d\}$, $J(2) = \{c,e\}$, and functions $J(\delta_-^-)$ the expected projections. Writing this up in table form we obtain:

0	1	2
a	b	c

0	1
a	b
a'	b

0	2
a	c
a'	c

1	2
b	c
d	e

0
a
a'

1
b
d

2
c
e

Strict relational instances, and strictification. Say that a relational instance is *tuplified* or *strict* if the keys are tuples and the δ's are (mapped to) projections . Accordingly, in a strict instance I on schema X we have that $I(x) \subseteq \prod_{A \in B_{\leq x}} I(A)$. (We reserve the product symbol $\prod$ for the dependent product in the category **Set** of sets. The dependent product in the type theory will be denoted by Π.) Thus Example D:8 is strict. It is clear that a relational instance is naturally isomorphic to exactly one strict relational instance with the same values. We say that the latter is the *tuplification* or *strictification* of the former.

Working with relational instances up to strictification, or restricting to the strict ones, resolves the coherence issues so typical of categorical models of type theory. To have the "strict" instances be those "on tuple form" presents itself as a natural choice, not least because of the connection to the relational model. Thus, formally we shall consider ourselves to work in the category of strict relational instances. We proceed informally, however, by allowing arbitrary relational instances but not distinguishing between relational instances that are equal up to tuplification.

Substitution. Let $f : X \longrightarrow Y$ be a morphism of schemas, and let $I : Y \longrightarrow \mathbf{FinSet}$ be a relational instance. Then it is easily seen that the composite $I \circ f : X \longrightarrow \mathbf{FinSet}$ is a relational instance. We write $I[f] := I \circ f$ and say it is the *substitution of I along f*.

Substitution does not preserve strict instances. If $f : X \longrightarrow Y$ is display, however, and $I : Y \longrightarrow \mathbf{FinSet}$ is strict, then we have that $I \circ f(x) \subseteq \prod_{A' \in B_{\leq f(x)}} \cong \prod_{A \in B_{\leq x}}$. The strictification of $I[f]$ is then just the reindexing of the tuples along the bijection $f_0 : B_{\leq x} \longrightarrow B_{\leq f(x)}$ (for all faces x). In contrast, if f is not display then there must exist a face, say $\{A, B\}$, such that $f(A) = f(B) = f(x)$, and then $f \circ I(x)$ need not be a set of tuples at all. (Accordingly, if we took an ordered perspective on schemas and morphisms and defined strict instances in terms of cartesian products, then display morphisms would be exactly the morphisms that preserve strict instances on the nose.) We display this for emphasis.

Lemma D:10. *Let $f : X \longrightarrow Y$ be a morphism of schemas. Then f is display if and only if substitution along f preserves instances on tuple form (up to reindexing).*

Example D:11. Consider the morphism $f : S \longrightarrow \Delta_2$ of Example D:4 and the instances I and J of Example D:6. Then $J[f]$ is the strictification of I, modulo the reindexing given by f_0.

The schema induced by an instance The connection between display maps, relational instances and simplicial schemas is given by the following. Let X be a schema and $F : X \longrightarrow \mathbf{FinSet}$ an arbitrary functor. Recall, e.g. from Mac Lane 1998, that the category of elements $\int_X F$ has objects $\langle x, a\rangle$ with $x \in X$ and $a \in F(x)$. A morphism $\delta^{\langle x,a\rangle}_{\langle y,b\rangle} : \langle x, a\rangle \longrightarrow \langle y, b\rangle$ is a morphism $\delta^x_y : x \longrightarrow y$ with $F(\delta^x_y)(a) = b$. The projection $p : \int_X F \longrightarrow X$ is defined by $\langle x, a\rangle \mapsto x$ and $\delta^{\langle x,a\rangle}_{\langle y,b\rangle} \mapsto \delta^x_y$. We then have

Lemma D:13. *Let X be a simplicial schema and $F : X \longrightarrow \mathbf{FinSet}$ be a functor. Then F is a relational instance if and only if $\int_X F$ is a simplicial schema and $p : \int_X F \to X$ is a display morphism.*

Proof. Let F be a relational instance. It is clear that there is at most one morphism between any two objects in $\int_X F$, and it is easy to see that $(\int_X F)^{\mathrm{op}}$ is a based poset with base $\{\langle A, a\rangle \mid A \in X_0,\ a \in F(A)\}$, satisfying the condition for being a simplicial complex. Furthermore, $(\int_X F)^{\mathrm{op}}_n = \{\langle x, c\rangle \mid x \in X_n,\ c \in F(x)\}$ so the projection $p : \int_X F \to X$ is a display morphism.

Conversely, if F is not relational, then the first condition for being a based poset is violated by any two keys with the same data. □

When F is a relational instance we write $X.F$ for $\int_X F$, and refer to it as the *canonical schema* of F. We refer to p as the *canonical projection*.

Example D:14. The instance J of Example D:6 has the canonical schema given by the attribute set $\{\langle 0, a\rangle, \langle 0, a'\rangle, \langle 1, b\rangle, \langle 1, d\rangle, \langle 2, e\rangle, \langle 2, c\rangle\}$ and relation variables e.g. $\langle\{0, 1, 2\}, \langle a, b, c\rangle\rangle$.

Terminal instances. A schema X induces a canonical instance of itself by filling out the relations by a single row each, consisting of the attributes of the relation. This instance is terminal in the category of instances of X; that is, every other instance of X has a unique morphism to it. It is of course isomorphic to the strict instance defined by $A \mapsto 1$, for $A \in X_0$ and $1 = \{*\}$ a fixed singleton set, and $x \mapsto ! : B_{\leq x} \longrightarrow 1$ for $x \in X \setminus X_0$. We take this (latter) instance to be the *terminal instance* $1_X : X \longrightarrow \mathbf{FinSet}$. For notational reasons, however, we allow ourself below to to think of it and write it as the functor defined by $x \mapsto \{x\}$.

Full tuples and induced sections. A *full* or *matching tuple* t of an instance I over schema X is a natural transformation $t : 1_X \Rightarrow I$. We write $\mathrm{Trm}_X(I)$ for the set of full tuples (indicating that we see them as terms type-theoretically). A full tuple t of a strict instance can be

considered as an element in $\prod_{A \in X_0} I(A)$ satisfying the condition that for all $x \in X$ we have that $t \restriction_{B_{\leq x}} \in I(x)$.

Given a full tuple $t : 1_X \Rightarrow I$, the *induced section* is the morphism $\hat{t} : X \longrightarrow X.I$ defined by $x \mapsto \langle x, t_x(x) \rangle$. Notice that the induced section is always a display morphism.

Example D:16. The instance I of Example D:6 has precisely two full tuples, one of which is given by $R \mapsto 1$, $Q \mapsto 1$, $A \mapsto a$, $B \mapsto b$, and $C \mapsto c$. A full tuple can be seen as a tuple over the full attribute set of the schema with the property that for all relation variables the projection of the tuple is a row of that relation. The two full tuples of I are, then, $\langle a, b, c \rangle$ and $\langle a', b, c \rangle$. The instance J of Example D:6 has precisely one full tuple $\langle a, b, c \rangle$.

2.2 Structure of the model

We have a functor $\mathrm{Rel}(-) : \mathbb{S}_d{}^{\mathrm{op}} \longrightarrow \mathrm{Cat}$ which maps X to $\mathrm{Rel}(X)$ and $f : X \longrightarrow Y$ to $\mathrm{Rel}(f) = (-)[f] : \mathrm{Rel}(Y) \longrightarrow \mathrm{Rel}(X)$. We denote this indexed category by $\mathfrak{R}$, and think of it as a "category of databases" in which the totality of databases and schemas are collected. It is a model of a certain dependent type theory with context constants which we give in Section 3. We briefly outline some of the relevant structure available in $\mathfrak{R}$.

Definition D:18. For $f : X \longrightarrow Y$ in $\mathbb{S}_d$ and $J \in \mathrm{Rel}(Y)$ and $t : 1_Y \Rightarrow J$ in $\mathrm{Trm}_Y(J)$:

1. Define $t[f] \in \mathrm{Trm}_X(J[f])$ by $x \mapsto t(f(x)) \in J[f](x)$. Note that $t[f][g] = t[f \circ g]$.

2. With $p_J : Y.J \longrightarrow Y$ the canonical projection, let $v_J : 1_{Y.J} \Rightarrow J[p_J]$ be the full tuple defined by $\langle y, a \rangle \mapsto a$. (We elsewhere leave subscripts on v and p determined by context.)

3. Denote by $\tilde{f} : X.J[f] \longrightarrow Y.J$ the schema morphism defined by $\langle x, a \rangle \mapsto \langle f(x), a \rangle$. Notice that since f is display, so is $\tilde{f}$.

Lemma D:19. *The following equations hold:*

1. *For X in $\mathbb{S}_d$ and $I \in \mathrm{Rel}(X)$ and $t \in \mathrm{Trm}_X(I)$ we have $p \circ \hat{t} = \mathrm{id}_X$ and $t = v[\hat{t}]$.*

2. *For $f : X \longrightarrow Y$ in $\mathbb{S}_d$ and $J \in \mathrm{Rel}(Y)$ and $t \in \mathrm{Trm}_Y(J)$ we have*

(a) $p \circ \tilde{f} = f \circ p : X.J[f] \longrightarrow Y$;
(b) $\tilde{f} \circ \widehat{t[f]} = \hat{t} \circ f : X \longrightarrow Y.J$; *and*
(c) $v_J[\tilde{f}] = v_{J[f]} : 1_{X.J[f]} \Rightarrow J[f][p]$.

3. *For* $f : X \longrightarrow Y$ *and* $g : Y \longrightarrow Z$ *in* $\mathbb{S}_d$ *and* $J \in \mathrm{Rel}(Z)$ *we have* $\widetilde{g \circ f} = \tilde{g} \circ \tilde{f}$.

4. *For* $X \in \mathbb{S}_d$ *and* $I \in \mathrm{Rel}(X)$ *we have* $\tilde{p} \circ \hat{v} = \mathrm{Id}_{X.I}$.

Proof. 1) $v[\hat{t}]$ is the full tuple of $J[p][\hat{t}] = J[p \circ \hat{t}] = J[\mathrm{id}_X]$ defined by $x \mapsto v(\hat{t}(x)) = v(\langle x, t(x)\rangle) = t(x)$. 2.a) we have $p \circ \tilde{f}(x,a) = p(f(x),a) = f(x) = f \circ p(x,a)$. 2.b) We have $\tilde{f} \circ \widehat{t[f]}(x) = \tilde{f}(x, t(fx)) = \langle f(x), t(fx)\rangle = \hat{t}(fx) = \hat{t} \circ f(x)$. 2.c) $v_J[\tilde{f}](x,a) = v_J(\widetilde{fx,a}) = a = v_{J[f]}(x,a)$. 3) We have $\tilde{g}(\tilde{f}(x,a)) = \tilde{g}(fx,a) = (gfx,a) = \widetilde{g \circ f}(x,a)$. 4) We have $\tilde{p} \circ \hat{v}(x,a) = \tilde{p}(\langle x,a\rangle, a) = \langle x,a\rangle$. □

Finally, we present the instance-forming operations of dependent product, dependent sum, 0 and 1, identity, and disjoint product.

Dependent product. Let $X \in \mathbb{S}$, $I \in \mathrm{Rel}(X)$, and $J \in \mathrm{Rel}(X.J)$. We define the instance $\Pi_J I : X \longrightarrow \mathbf{FinSet}$ as the right Kan-extension of J along p. Explicitly, we define the following strict instance (assuming also I and J strict).

For A in X_0 define

$$\Pi_J I(A) = \prod_{a \in I(A)} J(A,a)$$

Let $x \in X$, $f \in \prod_{A \in B_{\leq x}} \prod_{a \in I(A)} J(A,a)$, $y \leq x$, and $s \in I(y)$. Define

$$\hat{f}_{y,s} \in \prod_{\left\{\langle A,a\rangle \;\mid\; A \in B_{\leq y}, a = s(A)\right\}} J(A,a)$$

by $\langle A,a\rangle \mapsto f(A)(a))$. For $x \in X$ define

$$\Pi_I J(x) = \left\{ f \in \prod_{A \in B_{\leq x}} \prod_{a \in I(A)} J(A,a) \;\middle|\; \forall y \leq x.\ \forall s \in I(y).\ \hat{f}_{y,s} \in J(y,s) \right\}$$

Next, let $f \in \mathrm{Trm}_X(\Pi_I J)$ be a full tuple of the dependent product. We consider f as an element in $\prod_{A \in X_0} \Pi_I J(A)$ satisfying the condition that for all $x \in X$ we have that $f \restriction_{B_{\leq x}} \in \Pi_I J(x)$. Consider the element $\mathrm{Ap}_f \in \prod_{\langle A,a\rangle \in (X.I)_0} J(A,a)$ given by $\langle A,a\rangle \mapsto f(A)(a)$. Then for any $\langle y,s\rangle \in X.I$ we have $\mathrm{Ap}_f \restriction_{B_{\leq \langle y,s\rangle}} \in J(y,s)$ by the definition of $\Pi_J I(x)$, so $\mathrm{Ap}_f \in \mathrm{Trm}_{X.I}(J)$.

Finally, given $t \in \mathrm{Trm}_{X.I}(J)$, which we will consider as an element of $\prod_{\langle A,a\rangle \in (X.I)_0} J(A,a)$, define the element λt of $\prod_{A\in X_0} \Pi_I J(A)$ by $\lambda t(A)(a) = t(A,a)$. Then for all $x \in X$ we have that $\lambda t \restriction_{B_{\leq x}} \in \Pi_I J(x)$ since t is a full tuple.

Lemma D:20. *Let $f : X \longrightarrow\!\!\!\!\!\rightarrow Y$ be a display morphism in $\mathbb{S}$, $I \in \mathrm{Rel}(Y)$, $J \in \mathrm{Rel}(Y.I)$, $g \in \mathrm{Trm}_Y(\Pi_I J)$, and $t \in \mathrm{Trm}_{(Y.I)}(J)$.*

1. $\mathrm{Ap}_{\lambda t} = t$ *and* $\lambda \mathrm{Ap}_g = g$.
2. $(\Pi_I J)[f] = \Pi_{I[f]} J[\tilde{f}]$.
3. $(\lambda t)[f] = \lambda(t[\tilde{f}])$.
4. $\mathrm{Ap}_g[f] = \mathrm{Ap}_{g[f]}$.

Proof. Tedious but straightforward. □

Example D:21. Consider the schema S and instance I of Examples D:4 and D:6. Corresponding to the display map $f : S \longrightarrow\!\!\!\!\!\rightarrow \Delta_2$, we can present S an instance of Δ_2 as (ignoring strictification for readability) $S : \Delta_2 \longrightarrow \mathbf{FinSet}$ by $S(0) = \{A\}$, $S(1) = \{B\}$, $S(2) = \{C\}$, $S(01) = \{R\}$, $S(12) = \{Q\}$, and $S(02) = S(012) = \emptyset$. Notice that, modulo the isomorphism between S as presented in Example D:4 and $\Delta_2.S$, the morphism $f : S \longrightarrow\!\!\!\!\!\rightarrow \Delta_2$ is the canonical projection $p : \Delta_2.S \longrightarrow\!\!\!\!\!\rightarrow \Delta_2$. Similarly we have $I \in \Delta_2.S$ as (in tabular form, using subscript instead of pairing for elements in $\Delta_2.S$, and omitting the three single-column tables)

R_{01}	A_0	B_1
	a	b
	a'	b

Q_{12}	B_1	C_2
	b	c
	d	e

Then $\Pi_S I$ is, in tabular form,

	0	1	2
	a	b	c
	a'	b	c

	0	1
	a	b
	a'	b

	0	2
	a	c
	a'	c
	a	e
	a'	e

	1	2
	b	c
	d	e

0
a
a'

1
b
d

2
c
e

Notice that the three-column "top" table of $\Pi_S I$ is the natural join $R_{01} \bowtie Q_{12}$. The type theory of the next section will syntactically derive the rows of this table from the syntactic specification of S and I and the rules for the dependent product.

We present the remaining instance-forming operations more briefly. In particular, we omit the statements and (straightforward) proofs that all defined instances and terms are stable under substitution.

0 and 1 instances. Given $X \in \mathbb{S}$ the terminal instance 1_X has already been defined. The *initial* instance 0_X is the constant 0 functor, $x \mapsto \emptyset$. Note that $X.0_X$ is the empty schema.

Dependent sum. Let $X \in \mathbb{S}$, $I \in \mathrm{Rel}(X)$, and $J \in \mathrm{Rel}(X.I)$. We define the instance $\Sigma_I J : X \longrightarrow \mathbf{FinSet}$ (up to tuplification) by $x \mapsto \{\langle a, b\rangle \mid a \in I(x),\ b \in J(x, a)\}$. For δ^x_y in X, $\Sigma_I J(\delta^x_y)(a, b) = \left\langle \delta^x_y(a), \delta^{x,a}_{y,\delta^x_y(a)}(b) \right\rangle$.

Identity. Given $X \in \mathbb{S}$ and $J \in \mathrm{Rel}(X)$ the *Identity instance* $\mathrm{Id}_J \in \mathrm{Rel}(X.J.J[p])$ is defined, up to tuplification, by $\langle\langle x, a\rangle, b\rangle \mapsto 1$ if $a = b$ and $\langle\langle x, a\rangle, b\rangle \mapsto \emptyset$ else. The full tuple $\mathrm{refl} \in \mathrm{Trm}_{(X.J)}(\mathrm{Id}_J[\hat{v}])$ is defined by $\langle x, a\rangle \mapsto *$.

Disjoint union. Given $X \in \mathbb{S}$ and $I, J \in \mathrm{Rel}(X)$, the instance $I + J \in \mathrm{Rel}(X)$ is defined by $x \mapsto \{\langle n, a\rangle \mid \text{Either } n = 0 \wedge a \in I(x) \text{ or } n = 1 \wedge a \in J(x)\}$. We have full tuples $\mathrm{left} \in \mathrm{Trm}_{X.I}((I+J)[p])$ defined by $\langle x, a\rangle \mapsto \langle 0, a\rangle$ and $\mathrm{right} \in \mathrm{Trm}_{X.J}((I + J)[p])$ defined by $\langle x, a\rangle \mapsto \langle 1, a\rangle$.

3 The type theory

We introduce a Martin-Löf style type theory[35], with explicit substitutions (in the style of Dybjer 1996), extended with context and substitution constants representing simplices and face maps. The type theory contains familiar constructs such as Σ- and Π-types. It contains a universe which is closed under the other constructions, which we will describe in more detail in the next section. For this type theory we

[35]Martin-Löf 1984.

give an interpretation in the indexed category $\mathfrak{R}$ of the previous section. The goal is to use the type theory as a formal language for databases. We give examples how to specify instances and schemas formally in the theory, and remark on how to use the universe to talk about queries.

3.1 The type theory $\mathcal{T}$

The type system has the following eight judgements, with intended interpretations.

Judgement	Interpretation
$- : \mathtt{Context}$	$[\![-]\!]$ is a schema
$- : \mathtt{Type}(\Gamma)$	$[\![-]\!]$ is an instance of the schema Γ
$- : \mathtt{Elem}(A)$	$[\![-]\!]$ is an full tuple in the instance A
$- : \Gamma \longrightarrow \Lambda$	$[\![-]\!]$ is a (display) schema morphism
$\Gamma \equiv \Lambda$	$[\![\Gamma]\!]$ and $[\![\Lambda]\!]$ are equal schemas
$A \equiv B : \mathtt{Type}(\Gamma)$	$[\![A]\!]$ and $[\![B]\!]$ are equal instances of $[\![\Gamma]\!]$
$t \equiv u : \mathtt{Elem}(A)$	$[\![t]\!]$ and $[\![u]\!]$ are equal full tuples in $[\![A]\!]$
$\sigma \equiv \tau : \Gamma \longrightarrow \Lambda$	the morphisms $[\![\sigma]\!]$ and $[\![\tau]\!]$ are equal

The type theory $\mathcal{T}$ has the rules listed in figure 2. The interpretation of these are given by the constructions in the previous section, and summarised in figure 3.

Each rule introduces a context, substitution, type or element. We will apply usual abbreviations such as $A \to B$ for $\Pi_A B[\downarrow_A]$ and $A \times B$ for $\Sigma_A B[\downarrow_A]$. In addition to these term introducing rules there are a number of equalities which should hold; such as the simplicial identities $d_i^{n+1} \circ d_j^n \equiv d_{j-1}^{n+1} \circ d_i^n : \Delta_n \longrightarrow \Delta_{n+2}$. We list the definitional equalities in 4.

These all hold in our model. (The equalities for substitution are verified in Lemma D:19. The remaining equations are mostly routine verifications.) We display this for reference.

Theorem D:23. *The intended interpretation $[\![-]\!]$ yields a sound interpretation of the type theory $\mathcal{T}$ in $\mathfrak{R}$.*

3.2 Instance specification as type introduction

The intended interpretation of $\Gamma \vdash A\ \mathtt{type}$ is that A is an instance of the schema Γ. But context extension allows us to view every instance as a schema in its own right; for every instance $\Gamma \vdash A\ \mathtt{type}$, we get a schema $\Gamma.A$. It turns out that the most convenient way to specify a schema is by introducing a new type/instance over one of the simplex

schemas Δ_n. To specify a schema, with a maximum of n attributes, may be seen as introducing a type in the context Δ_n. A relation variable with k attributes in the schema is introduced as an element of the schema substituted into Δ_k. Names of attributes are given as elements of the schema substituted down to Δ_0.

Example D:24. We construct the rules of the schema S presented as an instance of Δ_2 as in Example D:21. The introduction rules tells us the names of tables and attributes in S.

$$
\begin{array}{ll}
S : \mathtt{Type}(\Delta_2) & A \equiv R[d_1] : \mathtt{Elem}(S[d_2 \circ d_1]) \\
A : \mathtt{Elem}(S[d_2 \circ d_1]) & B \equiv R[d_0] : \mathtt{Elem}(S[d_2 \circ d_0]) \\
B : \mathtt{Elem}(S[d_2 \circ d_0]) & B \equiv Q[d_1] : \mathtt{Elem}(S[d_0 \circ d_1]) \\
C : \mathtt{Elem}(S[d_0 \circ d_0]) & C \equiv Q[d_0] : \mathtt{Elem}(S[d_2 \circ d_0]) \\
R : \mathtt{Elem}(S[d_2]) & \\
Q : \mathtt{Elem}(S[d_0]) &
\end{array}
$$

From these introduction rules, we can generate an elimination rule. The elimination rule tells us how to construct full tuples in an instance over the schema S. Another interpretation of the elimination rule is that it formulates that the schema S contains only what is specified by the above introduction rules; it specifies the schema up to isomorphism.

$$
\begin{array}{l}
I : \mathtt{Type}(\Delta_2.S), a : \mathtt{Elem}(\Delta_0, I[(d_2 \circ d_1).S][A\uparrow]), \\
b : \mathtt{Elem}(\Delta_0, I[(d_2 \circ d_0).S][B\uparrow]), c : \mathtt{Elem}(\Delta_0, I[(d_0 \circ d_0).S][C\uparrow]), \\
r : \mathtt{Elem}(\Delta_1, I[d_2.S][R\uparrow]), q : \mathtt{Elem}(\Delta_1, I[d_0.S][Q\uparrow]), \\
r[d_1] \equiv a, r[d_0] \equiv b, q[d_1] \equiv b, q[d_0] \equiv c \\
\vdash \mathrm{rec}_S\ a\, b\, c\, r\, q : \mathtt{Elem}(\Delta_2.S, I)
\end{array}
$$

An instance of a schema is a type depending in the context of the schema. Therefore instance specification is analogous to schema specification.

Example D:26. Let S be the schema from the previous example. The following set of introductions presents an instance I of S.

$$
\begin{array}{ll}
\Delta_2.S \vdash I\ \mathtt{type} & \\
\vdash a : \mathtt{Elem}(\Delta_0, I[(d2 \circ d_1).S][A\uparrow]) & \vdash r_0[d_1] \equiv a \\
\vdash a' : \mathtt{Elem}(\Delta_0, I[(d2 \circ d_1).S][A\uparrow]) & \vdash r_0[d_0] \equiv b \\
\vdash b : \mathtt{Elem}(\Delta_0, I[(d2 \circ d_0).S][B\uparrow]) & \vdash r_1[d_1] \equiv a' \\
\vdash d : \mathtt{Elem}(\Delta_0, I[(d2 \circ d_0).S][B\uparrow]) & \vdash r_1[d_0] \equiv b \\
\vdash c : \mathtt{Elem}(\Delta_0, I[(d_0 \circ d_0).S][C\uparrow]) & \vdash q_0[d_1] \equiv b \\
\vdash e : \mathtt{Elem}(\Delta_0, I[(d_0 \circ d_0).S][C\uparrow]) & \vdash q_0[d_0] \equiv c \\
\vdash r_0 : \mathtt{Elem}(\Delta_1, I[(d2).S][R\uparrow]) & \vdash q_1[d_1] \equiv d \\
\vdash r_1 : \mathtt{Elem}(\Delta_1, I[(d2).S][R\uparrow]) & \vdash q_1[d_0] \equiv e \\
\vdash q_0 : \mathtt{Elem}(\Delta_1, I[(d_0).S][Q\uparrow]) & \\
\vdash q_1 : \mathtt{Elem}(\Delta_1, I[(d_0).S][Q\uparrow]) &
\end{array}
$$

The above is clearly very verbose, and can be compressed, at the cost of loosing control over the naming of attributes, into the following.

$$
\begin{array}{ll}
\vdash I : \mathtt{Type}(\Delta_2.S) & \\
\vdash r_0 : \mathtt{Elem}(\Delta_1, I[(d2).S][R\uparrow]) & \vdash r_0[d_0] \equiv r_1[d_0] \\
\vdash r_1 : \mathtt{Elem}(\Delta_1, I[(d2).S][R\uparrow]) & \vdash q_0[d_1] \equiv r_0[d_0] \\
\vdash q_0 : \mathtt{Elem}(\Delta_1, I[(d_0).S][Q\uparrow]) & \\
\vdash q_1 : \mathtt{Elem}(\Delta_1, I[(d_0).S][Q\uparrow]) &
\end{array}
$$

We omit the elimination rule.

4 Universe

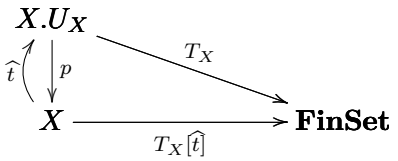
We construct the universe of finite instances of a schema. This is a large instance (in the sense of the small/large distinction of Section 2). Thus we allow in this section that schemas and instances can be large, that is, have infinitely many attributes, tables and rows.

4.1 Constructing the universe

An essential part of type theory is the notion of a universe of types. A universe of types is a family of types which is closed under some set of type constructors. This allows powerful reasoning about types in type theory itself. In this section we will, for each schema X, construct a universe of finite instances of X.

More precisely, A universe in type theory, in a context Γ, consists of a type $U : \mathtt{type}(\Gamma)$ and a family $T : \mathtt{type}(\Gamma.U)$. We think of the type U as the type of codes for types in the universe, and the family T as

decoding the codes into actual types (by substitution).



An important feature of a type theoretic universe is its closure under typeforming operations such as Π- and Σ-types. This is what allows reasoning about types internally in the type theory.

In order to encode the collection of finite instances into an instance of its own, we need a small set of tuples to work with. Let us therefore fix a set of values V closed under making lists, in the sense that $V^i \subseteq V$ for every finite subset $i \subseteq V$. This allows us to iterate tuple forming constructions such as Π and Σ.

Definition D:28. Define $D_n(V)$ to be the set of strict instances of the schema Δ_n which have values in V.

Definition D:29. Given a schema X, we define $U_X : X \to \text{Type}$ by

$$U_X(A) = \{\langle n, k, I\rangle \mid n \in \mathbb{N} \wedge k \in [n] \wedge I \in D_n(V)\}$$
for attributes A

$$U_X(x) = \{t : \prod_{A \in B_{\leq x}} U_X(A) \mid \pi_1 \circ t \text{ injective} \wedge \pi_0 \circ t \text{ and } \pi_2 \circ t \text{ constant}\}$$
for faces x

In order to define the decoding instance of the universe, which takes a code to its instance, we need easy access to the instances on attribute level.

Definition D:30. Given a schema X, a face x in X and a element $t \in U_X(x)$ we denote by $d(x,t) : \mathbb{N}$ the unique number such that $\pi_0(t(A)) = d(x,t)$ for all attributes A of x. Let $I(x,t)$ be the uniquely defined instance such that $\pi_2(t(A)) = I(x,t)$ for all attributes A of x. We denote by $\alpha(x,t) : \lfloor x \to \Delta_{d(x,t)}$ the morphism defined by $\alpha(x,t)(A) = \pi_1(tA)$ on attributes.

Lemma D:31. *Given $\sigma : X \to Y$ then for all $x \in X$ and $t \in U_X(x)$ we have that*

1. $d(\sigma(x), t \circ \sigma) = d(x,t)$

2. $I(\sigma(x), t \circ \sigma) = I(x,t)$

3. $\alpha(\sigma(x), t \circ \sigma) = \alpha(x,t) \circ \sigma$

Proof. 1. and 2. follows since $U_X(A) = U_X(\sigma(A))$ for all attributes. 3. is evident since $\alpha(\sigma(x), t \circ \sigma)(A) = \pi_1((t \circ \sigma)A) = \pi_1(t(\sigma(A)) = \alpha(x,t)(\sigma(A))$, by definition for all A □

Definition D:32. Given a schema X define $T_X : X.U_X \to \text{Type}$ by,

$$T_X(x,t) = I(x,t) \circ \alpha(x,t)$$

Proposition D:33. *The constructions U_- and T_- are invariant under substitution.*

Proof. U_- is invariant by construction; it is a strict instance which is constant on attributes.

T_- is invariant by the calculation, given $\sigma : X \to Y$

$$\begin{aligned} T_Y(\sigma(x), t \circ \sigma) &= I(\sigma(x), t \circ \sigma) \circ \alpha(\sigma(x), t \circ \sigma) \\ &= I(x,t) \circ \alpha(x,t) \circ \sigma \\ &= T_X(x,t) \circ \sigma \end{aligned}$$

□

Proposition D:34. *Given a schema X, a full tuple $a : 1 \Rightarrow U_X$ and a full tuple $b : 1 \Rightarrow U_{X.T_X[\hat{a}]}$ there is a full tuple $\sigma : 1 \Rightarrow U_X$ such that $T_X[\hat{\sigma}] = \Sigma_{T_X[\hat{a}]}(T_{X.T_X[\hat{a}]}[\hat{b}])$*

Proof. Let $Q = \Sigma_{T_X[\hat{a}]}(T_{X.T_X[\hat{a}]}[\hat{b}])$

For any face x of X, define an instance $S_x : \Delta_{d(x,a_x(*))} \to \text{Type}$

$$S_x(k) = \begin{cases} Q(x') & \text{where } x' \text{ is such that} \\ & \qquad \exists x''\ x \leq x'' \wedge x' \leq x'' \wedge \alpha(x', a_{x'}(*))(x') = k \\ \emptyset & \text{if no such } x' \text{ exists} \end{cases}$$

Let $\sigma_x(*)(A) = \langle d(x, a_x(*)), \alpha(x, a_x(*))(A), S_x \rangle$. This defines a term in U_X, since $\Sigma_{T_X[\hat{a}]}(T_{X[\hat{a}]}[\hat{b}])$ has values in V, and the definition of S_x is constant upwards and downwards in X.

Furthermore: $T_X[\hat{\sigma}] = \Sigma_{T_X[\hat{a}]}(T_{X[\hat{a}]}[\hat{b}])$, by construction.

□

Proposition D:35. *Given a schema X, a full tuple $a : 1 \Rightarrow U_X$ and a full tuple $b : 1 \Rightarrow U_{X.T_X[\hat{a}]}$ there is a full tuple $\pi : 1 \Rightarrow U_X$ such that $T_X[\hat{\pi}] = \Pi_{T_X[\hat{a}]}(T_{X.T_X[\hat{a}]}[\hat{b}])$*

Proof. Let $Q = \Pi_{T_X[\hat{a}]}(T_{X.T_X[\hat{a}]}[\hat{b}])$

For any face x of X, define an instance $S_x : \Delta_{d(x,a_x(*))} \to \mathrm{Type}$

$$S_x(k) = \begin{cases} Q(x') & \text{where } x' \text{ is such that} \\ & \qquad \exists x''\ x \leq x'' \wedge x' \leq x'' \wedge \alpha(x', a_{x'}(*))(x') = k \\ \emptyset & \text{if no such } x' \text{ exists} \end{cases}$$

Let $\pi_x(*)(A) = \langle d(x, a_x(*)), \alpha(x, a_x(*))(A), S_x \rangle$. This defines a term in U_X, since $\Pi_{T_X[\hat{a}]}(T_{X[\hat{a}]}[\hat{b}])$ has values in V, and the definition of S_x is constant upwards and downwards in X.

Furthermore: $T_X[\hat{\pi}] = \Pi_{T_X[\hat{a}]}(T_{X[\hat{a}]}[\hat{b}])$, by construction.

□

4.2 Using the universe in the type theory

The rules for the universe in the type theory it self can be summarized as

$$\Gamma : \mathtt{Context} \vdash U : \mathtt{Type}(\Gamma)$$
$$\Gamma : \mathtt{Context} \vdash T : \mathtt{Type}(\Gamma.U),$$

along with rules for invariance under substitution, and closure rules for each type-formation rule we previously had. For instance, the closure rule for Π is the following:

$$\begin{array}{ll} \Gamma : \mathtt{Context} & \vdash \pi_\Gamma : \mathtt{Elem}(U[\downarrow_U][\downarrow_{T \twoheadrightarrow U}]) \\ a : \mathtt{Elem}(U), & \\ f : \mathtt{Elem}(T[a\uparrow] \twoheadrightarrow U) & \vdash \pi[a\uparrow.(T \twoheadrightarrow U)][f\uparrow] \equiv \Pi_{T[a\uparrow]} T[\mathrm{apply}_f \uparrow] : \mathtt{Type}(\Gamma) \end{array}$$

The universe, $U_\Gamma : \mathtt{Type}(\Gamma)$ along with $T_\Gamma : \mathtt{Type}(\Gamma.U_\Gamma)$, allows reasoning generically about classes of instances of Γ in the type theory itself, without having to resort to the metalanguage. Since schemas can be though of as instances, they too can be constructed using the universe. In particular, given a schema Γ, the type $\Omega_\Gamma := \Sigma_{U_\Gamma} \Pi_{T_\Gamma} \Pi_{T_\Gamma[\downarrow_{T_\Gamma}]} \mathrm{Id}_{T_\Gamma}$ is the large type of subschemas of Γ. Its elements are decoded to instances

by the family $\mathcal{O}_\Gamma := T[\downarrow_\Omega .U][\pi_0\uparrow]$. Given $t : \mathtt{Elem}(\Omega_\Gamma)$, the subschema it encodes is $\Gamma.\mathcal{O}[t\uparrow]$.

A query can then be seen as an operation which takes an instance of a schema to another instance of a related schema. Given codes for a source subschema $t : \mathtt{Elem}(\Omega)$ and a target subschema $u : \mathtt{Elem}(\Omega)$, the type of queries from t to u is thus $(\mathcal{O}[t\uparrow] \twoheadrightarrow U) \twoheadrightarrow (\mathcal{O}[u\uparrow] \twoheadrightarrow U)$. Having given a concrete type of queries leads the way to investigations as to exactly which queries can be expressed in the language. For illustration, we present an example query formulated in this way.

Example D:36. In the spririt of example D:21, let $a : \mathtt{Elem}(\Omega)$ be the code for a subschema covering the schema Γ, in the sense that the set of attributes are the same. The query taking the dependent product, or natural join, of an instance of this subschema is expressed by the term

$$q = \lambda\lambda(\pi[(\downarrow_\Omega .U) \circ (\pi_0\uparrow)][a\uparrow.(T \twoheadrightarrow U)][\downarrow_1]) : \mathtt{Elem}((\mathcal{O}[a\uparrow] \twoheadrightarrow U) \twoheadrightarrow (1_\Gamma \twoheadrightarrow U)).$$

5 Representing data simplicially

We collect in this section some further examples of tables of data organized "simplicially", and discuss briefly certain aspects of this representation. The sparse examples of Section 2 were meant to illustrate the technical definitions, and were kept rather to a minimum. To begin with, the following examples give some further illustrations of context extension and Σ and Π-types. (We exploit in these examples, to simplify them a bit, the rather immediate back-and-forth translation between "simplicial" schemas/instances and the traditional schemas/instances of the relational model. We omit part-relation tables when they can be assumed to be projections, for instance. We also make some cosmetic simplifications such as not necessarily writing the attributes of induced schemas as pairs.)

Example D:38. Let S be the schema and I the instance as follows:

State	Head of state
Monarchy	King
Monarchy	Queen
Republic	President

We write $S \vdash I$ for "S is a schema and I is an instance over S". Then the schema $S.I$ induced by I (p. 113) is:

Monarchy	King

Monarchy	Queen

Republic	President

Now let $S.I \vdash J$ be defined as follows:

Monarchy	Queen
United Kingdom	Elizabeth II
Denmark	Margrethe II

Monarchy	King
Norway	Harald V
Sweden	Carl XVI Gustav

Republic	President
Finland	Sauli Niinistö
Iceland	Ólafur Ragnar Grímsson

The dependent sum $S \vdash \Sigma_I J$ (p. 117) is then the instance collecting all the tables into the table of the original schema S:

State	Head of state
⟨ Monarchy, Norway ⟩	⟨ King, Harald V ⟩
⟨ Monarchy, Sweden ⟩	⟨ King, Carl XVI Gustav ⟩
⟨ Monarchy, United Kingdom ⟩	⟨ Queen, Elizabeth II ⟩
⟨ Monarchy, Denmark ⟩	⟨ Queen, Margretha II ⟩
⟨ Republic, Finland ⟩	⟨ President, Sauli Niinistö ⟩
⟨ Republic, Iceland ⟩	⟨ President, Ólafur Ragnar Grímsson ⟩

Example D:40. Let $R \vdash K$ be the schema and instance

Vehicle	Type	Wheels

Vehicle	Type
Vehicle	Type

Vehicle	Wheels

Type	Wheels
Type	Wheels

K is a "subsingleton" or "subterminal" instance. It has not more than one row in every table, and is thereby a subinstance of the terminal instance 1_R wich has excactly one row in each table (p. 113). Accordingly, K can be regarded as a subschema of R.

We form the induced schema $R.K$. Let $R.K \vdash L$ be the following instance:

Vehicle	Type
Ford Model T	Car
Triumph Tiger 100	Motorcycle
Peugeot Type 3	Car

Type	Wheels
Car	4
Motorcycle	2

The dependent product $R \vdash \Pi_A B$ (p. 115) can in this case be computed by instantiating the tables given by R in ascending order by taking the natural join of the tables in L lying (not necessarily properly) below it; thus the attribute level tables are those of L

Vehicle
Ford Model T
Triumph Tiger 100
Peugeot Type 3

Type
Car
Motorcycle

Wheels
2
4

Similarly, the [Vehicle | Type] and [Type | Wheels] tables are those of L. While the [Vehicle | Wheels] table, for which L provides no information, is the natural join of [Vehicle] and [Wheels]

Vehicle	Wheels
Ford Model T	4
Ford Model T	2
Triumph Tiger 100	4
Triumph Tiger 100	2
Peugeot Type 3	4
Peugeot Type 3	2

Finally, [Vehicle | Type | Wheels] is the natural join of [Vehicle | Type], [Type | Wheels], and [Vehicle | Wheels] as just given, thus:

Vehicle	Type	Wheels
Ford Model T	Car	4
Triumph Tiger 100	Motorcycle	2
Peugeot Type 3	Car	4

Example D:42. To give an example of the dependent product where the "middle" instance is not a subschema, we can compute $S \vdash \Pi_I J$ for $S.I \vdash J$ of Example D:38. Unfortunately, J has no full tuples; no choice of values for the attributes Monarchy-Republic-King-Queen-President yields a full tuple, as there are no monarchies in J with both a queen and a king head of state. Thus $S \vdash \Pi_I J$ becomes:

State	Head of state

State
⟨ Denmark, Finland ⟩
⟨ Denmark, Iceland ⟩
⟨ Norway, Finland ⟩
⟨ Norway, Iceland ⟩
⟨ Sweden, Finland ⟩
⟨ Sweden, Iceland ⟩
⟨ United Kingdom, Finland ⟩
⟨ United Kingdom, Iceland⟩

Head of state
⟨Elizabeth, Harald , Niinistö⟩
⟨Elizabeth, Harald , Grímsson⟩
⟨Elizabeth, Carl Gustav , Niinistö⟩
⟨Elizabeth, Carl Gustav , Grímsson⟩
⟨Margrethe, Harald, Niinistö⟩
⟨Margrethe, Harald , Grímsson⟩
⟨Margrethe, Carl Gustav , Niinistö⟩
⟨Margrethe, Carl Gustav , Grímsson⟩

But if we let J be, instead

Monarchy	Queen
Swaziland	Ndlovukati
United Kingdom	Elizabeth II
Denmark	Margrethe II

Monarchy	King
Swaziland	Ngwenyama
Norway	Harald V
Sweden	Carl XVI Gustav

Republic	President
Finland	Sauli Niinistö
Iceland	Ólafur Ragnar Grímsson

Then the State–Head of state table of $\Pi_I J$ becomes

State	Head of state
⟨ Swaziland, Finland ⟩	⟨Ndlovukati, Ngwenyama , Niinistö⟩
⟨ Swaziland, Iceland ⟩	⟨Ndlovukati, Ngwenyama, Grímsson⟩

representing the two full tuples of J.

We proceed with an example suggesting the use of context extension — the built in possibility to enter data related to the instance into tables formed by its rows+,— for updates; an update I' of an instance I over S is the instance over $S.I$ obtained by writing the new (or old or empty) row in the table formed by the row to be replaced (or kept or deleted). Adding new rows can be done by writing I' over $S.I+1$ instead, as $I+1$ has a copy of S over which new additions can be entered. (Multiple copies of S, and indeed of I, can be added if need be; notice that polynomial expressions over I such as $2I+3$ yield meaningful instances over S). In this way a current update occurs in a context formed by a string of previous updates, thus displaying provenance. Applying the dependent product operation gives an instance over the original schema S, if desired. (In the following example we return to writing the attributes of induced schemas as attribute-value pairs. For esthestic reasons we write pairs as A:a, and we airbrush away the pairs with 0 and 1 used for the elements of the disjoint union.)

Example D:44. Let $P \vdash M$ be the schema and instance:

	First name(s)	Last name	SSN	Department
	Jim T.	Kirk	333	Astrophysics
	James	Moriarty	222	Criminology

$P \vdash M+1$ (p. 117) adds a dummy row:

	First name(s)	Last name	SSN	Department
	Jim T.	Kirk	333	Astrophysics
	James	Moriarty	222	Criminology
	$\star$	$\star$	$\star$	$\star$

An update which corrects Kirk's first name, deletes Moriarty, and inserts two new rows can be written as an instance $P.(M+1) \vdash N$ (We abbreviate some of the strings involved):

	FN: Jim T.	LN: Kirk	SSN: 333	Dep: Astro
	James Tiberius	Kirk	333	Astrophysics

	FN: James	LN: Moriarty	SSN: 222	Dep: Crime

	FN: ⋆	LN: ⋆	SSN: ⋆	Dep: ⋆
	Sherlock	Holmes	111	Criminology
	James	Watson	555	Medicin

The dependent sum $P \vdash \Sigma_{M+1} N$, collecting this over the original schema, becomes:

	First name(s)	Last name	SSN	Department
	Jim T. : James Tiberius	Kirk : Kirk	333 : 333	Astro : Astro
	⋆ : Sherlock	⋆ : Holmes	⋆ : 111	⋆ : Crime
	⋆ : James	⋆ : Watson	⋆ : 555	⋆ : Med

The one condition on simplicial schemas that can hinder a straightforward translation from a relational schema is the condition that there can be at most one relation variable over a given set of attributes. In cases where there is a conflict and it is unnatural to rename attributes, this will have to be worked around, for instance by adding extra attributes to keep tables distinct.

Example D:46. Faculty and staff tables, both with intended attributes First name, Last name, SSN, and Department. Separated with table name attribute with dummy value:

	Staff	First name(s)	Last name	SSN	Department
	⋆	Jim T.	Kirk	333	Astrophysics
	⋆	James	Moriarty	222	Criminology

	Faculty	First name(s)	Last name	SSN	Department
	⋆	Khan N.	Sing	666	Astrophysics
	⋆	Sherlock	Holmes	111	Criminology

Separated with primary key columns:

	Staff_Id	First name(s)	Last name	SSN	Department
	1	Jim T.	Kirk	333	Astrophysics
	2	James	Moriarty	222	Criminology

Faculty_Id	First name(s)	Last name	SSN	Department
1	Khan N.	Sing	666	Astrophysics
2	Sherlock	Holmes	111	Criminology

In one table:

Staff/Faculty	First name(s)	Last name	SSN	Department
Staff	Jim T.	Kirk	333	Astrophysics
Staff	James	Moriarty	222	Criminology
Faculty	Khan N.	Sing	666	Astrophysics
Faculty	Sherlock	Holmes	111	Criminology

Finally, we illustrate the suggestion of using part tables to record missing information.

Example D:48. Holmes and Watson have unknown Social Security Numbers. Watson has not been assigned a department.

First name(s)	Last name	SSN	Dep
Jim T.	Kirk	333	Astroph

First name(s)	Last name	Dep
Jim T.	Kirk	Astroph
Sherlock	Holmes	Crimin

First name(s)	Last name
Jim T.	Kirk
Sherlock	Holmes
James	Watson

all other part tables are projections.

References

Abiteboul, Serge, Richard Hull, and Victor Vianu (1995). *Foundations of Databases*. Addison-Wesley, Reading MA.

Cartmell, John (1986a). "Formalising the network and hierarchical data models — an application of categorical Logic". English. In: *Category Theory and Computer Programming*. Ed. by David Pitt et al. Vol. 240. LNCS. Springer, Heidelberg, pp. 466–492. ISBN: 978-3-540-17162-1. DOI: 10.1007/3-540-17162-2_138.

— (1986b). "Generalised algebraic theories and contextual categories". In: *Annals of Pure and Applied Logic* 32, pp. 209–243. ISSN: 0168-0072. DOI: http://dx.doi.org/10.1016/0168-0072(86)90053-9. URL: http://www.sciencedirect.com/science/article/pii/0168007286900539.

Dybjer, Peter (1996). "Internal type theory". English. In: *Types for Proofs and Programs*. Ed. by Stefano Berardi and Mario Coppo. Vol. 1158. LNCS. Springer, Heidelberg, pp. 120–134. ISBN: 978-3-540-61780-8. DOI: 10.1007/3-540-61780-9_66.

Friedman, Greg (2012). "Survey Article: An elementary illustrated introduction to simplicial sets". In: *Rocky Mountain J. Math.* 42.2, pp. 353–423. DOI: 10.1216/RMJ-2012-42-2-353.

Gabriel, Peter and Michel Zisman (1967). *Calculus of Fractions and Homotopy Theory*. Springer, Heidelberg.

Grandis, Marco (2001). "Finite Sets and Symmetric Simplicial Sets". In: *Theory and Applications of Catefories* 8.8, pp. 244–252.

Jacobs, Bart (1999). *Categorical Logic and Type Theory*. Elsevier, Amsterdam.

Mac Lane, Saunders (1998). *Categories for the Working Mathematician*. Springer, Heidelberg.

Martin-Löf, Per (1984). *Intuitionistic type theory. Notes by Giovanni Sambin*. Vol. 1. Studies in Proof Theory. Bibliopolis, Naples, pp. iv+91. ISBN: 88-7088-105-9.

Spivak, David (2009). *Simplicial Databases*. URL: http://arxiv.org/abs/0904.2012.

Figure 2: Rules of the type theory

$\sigma : \Gamma \longrightarrow \Delta,\ \tau : \Delta \longrightarrow \Theta$	$\vdash \tau \circ \sigma : \Gamma \longrightarrow \Theta$
$\Gamma : \mathtt{Context}$	$\vdash \mathrm{id}_\Gamma : \Gamma \longrightarrow \Gamma$
$A : \mathtt{Type}(\Gamma),\ \sigma : \Delta \longrightarrow \Gamma$	$\vdash A[\sigma] : \mathtt{Type}(\Gamma)$
$a : \mathtt{Elem}(A),\ \sigma : \Delta \longrightarrow \Gamma$	$\vdash a[\sigma] : \mathtt{Elem}(A[\sigma])$
$A : \mathtt{Type}(\Gamma)$	$\vdash \Gamma.A : \mathtt{Context}$
$A : \mathtt{Type}(\Gamma)$	$\vdash \downarrow_A : \Gamma.A \longrightarrow \Gamma$
$A : \mathtt{Type}(\Gamma)$	$\vdash v : \mathtt{Elem}(A[\downarrow_A])$
$a : \mathtt{Elem}(A)$	$\vdash a\uparrow : \Gamma \longrightarrow \Gamma.A$
$A : \mathtt{Type}(\Gamma),\ \sigma : \Delta \longrightarrow \Gamma$	$\vdash \sigma.A : \Delta.A[\sigma] \longrightarrow \Gamma.A$
$A : \mathtt{Type}(\Gamma),\ B : \mathtt{Type}(\Gamma.A)$	$\vdash \Pi_A B : \mathtt{Type}(\Gamma)$
$b : \mathtt{Elem}(B)$	$\vdash \lambda b : \mathtt{Elem}(\Pi_A B)$
$f : \mathtt{Elem}(\Pi_A B)$	$\vdash \mathrm{apply}_f : \mathtt{Elem}(B)$
$A : \mathtt{Type}(\Gamma),\ B : \mathtt{Type}(\Gamma.A)$	$\vdash \Sigma_A B : \mathtt{Type}(\Gamma)$
$A : \mathtt{Type}(\Gamma),\ B : \mathtt{Type}(\Gamma.A)$	$\vdash \mathrm{pair} : \mathtt{Elem}(\Sigma_A B[\downarrow_A][\downarrow_B])$
$C : \mathtt{Type}(\Gamma.\Sigma_A B),$	
$c_0 : \mathtt{Elem}(C[(\downarrow_A \circ \downarrow_B).\Sigma_A B][\mathrm{pair}\uparrow])$	$\vdash \mathrm{rec}_\Sigma\ c_0 : \mathtt{Elem}(C)$
$A : \mathtt{Type}(\Gamma)$	$\vdash \mathrm{Id}_A : \mathtt{Type}(\Gamma.A.A[\downarrow_A])$
$A : \mathtt{Type}(\Gamma)$	$\vdash \mathrm{refl} : \mathtt{Elem}(\mathrm{Id}_A[v\uparrow])$
$C : \mathtt{Type}(\Gamma.A.A[\downarrow_A].\mathrm{Id}_A),$	
$c_0 : \mathtt{Elem}(C[(v\uparrow).\mathrm{Id}_A][\mathrm{refl}\uparrow])$	$\vdash \mathrm{rec}_{\mathrm{Id}}\ c_0 : \mathtt{Elem}(C)$
$A : \mathtt{Type}(\Gamma)\ B : \mathtt{Type}(\Gamma)$	$\vdash A + B : \mathtt{Type}(\Gamma)$
$A : \mathtt{Type}(\Gamma)\ B : \mathtt{Type}(\Gamma)$	$\vdash l_{A,B} : \mathtt{Elem}((A + B)[\downarrow_A])$
$A : \mathtt{Type}(\Gamma)\ B : \mathtt{Type}(\Gamma)$	$\vdash r_{A,B} : \mathtt{Elem}((A + B)[\downarrow_B])$
$C : \mathtt{Type}(\Gamma.(A + B)),$	
$c_0 : \mathtt{Elem}(C[(\downarrow).(A + B)][l_{A,B}])$	
$c_1 : \mathtt{Elem}(C[(\downarrow).(A + B)][r_{A,B}])$	$\vdash \mathrm{rec}_+\ c_0\, c_1 : \mathtt{Type}(C)$
$\Gamma : \mathtt{Context}$	$\vdash 0 : \mathtt{Type}(\Gamma)$
$A : \mathtt{Type}(\Gamma.0)$	$\vdash \mathrm{rec}_0 : \mathtt{Elem}(A)$
$\Gamma : \mathtt{Context}$	$\vdash 1 : \mathtt{Type}(\Gamma)$
$\Gamma : \mathtt{Context}$	$\vdash * : \mathtt{Elem}(1)$
$A : \mathtt{Type}(\Gamma.1), a : A[*\uparrow]$	$\vdash \mathrm{rec}_1\, a : \mathtt{Elem}(A)$
	$\vdash \Delta_n : \mathtt{Context}$
	$\vdash d_i^n : \Delta_n \longrightarrow \Delta_{n+1}$

where $n, i, j \in \mathbb{N}$ are such that $i < j \leq n + 2$.

Figure 3: Interpretation of the type theory

$$
\begin{array}{cccc}
[\![\sigma \circ \tau]\!] = [\![\sigma]\!] \circ [\![\tau]\!] & [\![A[\sigma]]\!] = [\![A]\!]\,[[\![\sigma]\!]] & [\![id_\Gamma]\!] = id_{[\![\Gamma]\!]} & [\![a[\sigma]]\!] = [\![a]\!]\,[[\![\sigma]\!]] \\
[\![\Gamma.A]\!] = [\![\Gamma]\!].\,[\![A]\!] & [\![\downarrow_A]\!] = p & [\![v]\!] = v & [\![a\uparrow]\!] = \widehat{[\![a]\!]} \\
[\![\sigma.A]\!] = \widetilde{[\![\sigma]\!]} & [\![\Pi_A B]\!] = \Pi_{[\![A]\!]}\,[\![B]\!] & [\![\lambda f]\!] = \lambda\,[\![f]\!] & [\![\mathrm{apply}]\!] = Ap \\
[\![\Sigma_A B]\!] = \Sigma_{[\![A]\!]}\,[\![B]\!] & [\![\mathrm{pair}]\!] = \mathrm{pair} & [\![\mathrm{rec}_\Sigma]\!] = \mathrm{rec}_\Sigma & [\![\mathrm{Id}_A]\!] = \mathrm{Id}_{[\![A]\!]} \\
[\![\mathrm{refl}]\!] = \mathrm{refl} & [\![\mathrm{rec}_{\mathrm{Id}}]\!] = \mathrm{rec}_{\mathrm{Id}} & [\![\Delta_n]\!] = \Delta_n & [\![d_i^n]\!] = d_i^n
\end{array}
$$

Figure 4: Definitional equalities in the type theory

$$
\begin{array}{ccc}
(\upsilon \circ \tau) \circ \sigma \equiv \upsilon \circ (\tau \circ \sigma) & id_\Gamma \circ \sigma \equiv \sigma & \sigma \circ id_\Gamma \equiv \sigma \\
a[\sigma \circ \tau] \equiv a[\sigma][\tau] & a[id_\Gamma] \equiv a & \downarrow_A \circ (a\uparrow) \equiv id_\Gamma \\
v[a\uparrow] \equiv a & \downarrow_A \circ (\sigma.A) \equiv \sigma \circ \downarrow_{A[\sigma]} & v[f.A] \equiv v \\
(\sigma.A) \circ (a[\sigma]\uparrow) \equiv a\uparrow \circ \sigma & (\sigma.A) \circ (\tau.A[\sigma]) \equiv (\sigma \circ \tau).A & \downarrow .A \circ v\uparrow \equiv Id_{\Gamma.A} \\
\Pi_A B[\sigma] \equiv \Pi_{A[\Sigma]} B[\sigma.A] & \mathrm{apply}(\lambda b) \equiv b &
\end{array}
$$

E

Dependent Term Systems

Abstract

We consider two approaches to unravelling the dependency structures between terms in dependent type theory. The first approach defines a notion closely related to Cartmell's categories with attributes, but with a separate operations for extending a context with a defined term. The second approach is closely related to Makkai's one-way categories — which form the signatures of his First-Order Logic with Dependent Sorts (FOLDS). This preliminary investigation provides the definitions and some basic results for both approaches. We also formulate a few conjectures about how these two approaches relate to guide further investigation.

1 Introduction

Dependently typed theories, such as Martin-Löf's Intentional Type Theory (ITT), are powerful tools in logic and computer science. Their power lies in their expressivity and in their varied semantics. A recent surge of interest in the topic comes from a string of results relating ITT to homotopy theory. One side of this relation is the construction of models of ITT based on various notions of spaces in homotopy theory, such as cubical sets. Related to this is the formulation of new type theories, with new forms of judgements inspired by these models - in particular Cubical Type Theory (CTT).

For their popularity, it is not easy to pin down exactly what constitutes *a dependently typed theory*, or in general what a model of such a thing is. This is the end goal of several current research projects.

When it comes to modelling ITT, there are many variants of the basic theory — many of which are equivalent. We here mention

- Categories with Attributes (CwAs)
- Categories with Families (CwFs)
- Comprehension categories (CompCats)

These are all variations of setting up a framework of contexts, substitutions and types. However, these notions need to be extended for each rule of the type theory. Thus one speaks of CwAs with Π-structure, Σ-structure, etc.

A different flavour of dependent types are modelled by Makkai's *one-way categories*, which are the categorical equivalent of well-founded posets. One-way categories provide a simple model of dependent sorts, but say nothing of terms of these sorts. In his development of First Order Logic With Dependent Sorts (FOLDS), which uses one-way categories as their signatures, Makkai 1995 works around the lack of terms by coding them using relations in fruitful ways. Belo 2008 and Palmgren 2016 have each developed a syntactic notion of dependently sorted system with terms.

In this work, we set out to find a semantic notion of term-signatures for one-way categories. We see this as a further step towards reconciling the two notions of dependencies, the one represented by CwAs and the other represented by one-way categories.

2 Notation

In our mathematical descriptions, we will use the follow notations.

- `Set` refers to the collection of sets.
- `x : A` will denote `x` is a member of the collection or set `A`.
- $\sum$`(`$\mathbb{C}$`,F)` denotes the (covariant) Grothendieck construction applied to $\mathbb{C}$ `: Cat` and `F :` $\mathbb{C} \to$ `Set`.
- `f x` denotes the value of the function `f` applied to `x`.
- `f(A)` denotes the image of `A` by `f`, as a subset of the codomain of `f`, whenever `A` is a subset of the domain of `f`.
- `A - x` denotes the subset of `A` of elements not equal to `x`.
- `x` $\in$ `A` will denote that `x` is a member of the subset `A`. Which set `A` is a subset of is left implicit.
- `A` $\subseteq$ `B` means either
  - `A` is a subset of `B`,
  - or that each element of the subset `A` is also an element of the subsets `B`. Again the set of which `A` and `B` are subsets of is left implicit.
- `f : x` $\to$ `y` denotes various kinds of morphisms, usually disambiguated by a noun phrase before it. Example: "a functor `F : C` $\to$ `D`".
- α `: F` $\Rightarrow$ `G` denotes that α is a natural transformation from `F` to `G`, for two parallel functors `F,G : C` $\to$ `D`.
- α `x` denotes the `x` component of a natural transformation α `: F` $\to$ `G` for `x : Ob C`, where `F,G : C` $\to$ `D`.

3 Categories with attributes and definitions

We will introduce the notion of categories with definitions, which is a slight variation of the notion of categories with attributes (CwA). In fact, one can see categories with definitions as categories with attributes equipped with a bit of extra structure. For the reader already familiar with categories with attributes, we may summarise categories with definitions as extending categories with attributes with an operation for extending contexts with defined terms. This extension mechanism allows reasoning about dependencies between terms.

One motivation for this notion comes from computer science. Interpreters for programming languages often need to keep track of what is called *an environment* — a set of names, each equipped with either a declaration that it is a variable, or a definition if it is a defined term. When a name is encountered in a term, the interpreter looks it up in the environment to see if it is defined or a variable, and may replace it by its definition in the course of evaluation. Thus, we would like a formalism which captures the notion that contexts include not only a list of assumptions, but also a set of terms thus far constructed and defined.

Categories with definitions appeals also to our constructive instincts, where our model mathematician may not encounter infinitely many terms at once, but keeps a finite number of them in his mind or on paper, and considers at each point a finite number of new possible terms which he can extend his investigation with. A category with definitions may model this by only making a finite number of terms available in each context, and adding more terms only once some of the previous terms have been defined. It will also make the distinction between the name of a defined term and the term it self.

One thing to note, is that we will consider arrows to go in the opposite direction of what is usually done in CwAs. That is, we will study *covariant CwAs*, as opposed to the standard, contravariant ones. For example, we will have an "inclusion morphism" $\iota\,\mathtt{A} : \Gamma \to \Gamma.\mathtt{A}$, instead of a "projection morphism" $\mathtt{p} : \Gamma.\mathtt{A} \to \Gamma$. The reason is that we want to see contexts, as collections of names[36], and morphisms should be mapping between those names. The context $\Gamma.\mathtt{A}$ has all the names of Γ, plus a new name for a variable of type $\mathtt{A}$. Thus there is naturally an inclusion of the names of Γ to the names of $\Gamma.\mathtt{A}$.

[36]By *name*, we will mean a way to reference a term. In CwAs there is no distinction between a term and a name, but categories with definitions will have such a distinction.

3.1 Categories with Attributes

Categories with attributes were introduced in Cartmell 1986, and have been successfully used to give semantics of intensional type theory. One of the more celebrated results is that the syntax of type theory gives an initial CwA[37], and thus models can be constructed as morphisms from the syntactic CwA to another CwA. We will defer a precise definition of CwAs a bit, and instead give an account of CwAs based on the intuition of contexts as collections of variables (and terms).

A category with attributes is a category $\mathbb{C}$, with some extra structure on it. We think of the objects Γ `:` `Ob` $\mathbb{C}$ as contexts — in the syntactic case, sequences of typed assumptions — which form the basis of our types and terms. The CwA comes with a functor `Ty` $: \mathbb{C} \to$ `Set` which for each context gives us the types available in that context. For instance if `(n:`$\mathbb{N}$`)` represents the context assuming a single natural number, then we might have `Vec : Ty (n:`$\mathbb{N}$`)` representing the type of vectors of length `n`. Conceptually, `Vec` is a type depending on the type $\mathbb{N}$. Usually, one assumes that there is an empty context `() : Ob` $\mathbb{C}$, which is an initial object — but it is not essential.

The fact that `Ty` is a functor means that we can substitute types along context morphisms. As mentioned, we let substitution run forward in the direction of the arrows in $\mathbb{C}$, so *`Ty` will be a covariant functor*. Concretely, if `A : Ty` Γ and $\sigma : \Gamma \to \Delta$ then `Ty` σ `A`, often denoted `A[`σ`]`, is a type in Δ.

Once a type in a context is given, say `A : Ty` Γ, we can further extend the context with a new variable of that type — which is denoted Γ`.A`. The types in context Γ`.A` are intended to be types depending on a variable in `A`. There is a morphisms giving the weakening substitution $\iota : \Gamma \to \Gamma$`.A`. The context extension operation is functorial, so we can lift morphisms $\sigma : \Gamma \to \Delta$ to σ`.A` $: \Gamma$`.A` $\to \Delta$`.A[`σ`]`. Furthermore, ι has a certain naturality with respect to this lifting — and the associated naturality square is a push-out diagram.

Types are no fun without terms, but in CwA's one does not usually formulate terms explicitly as a functor, like one does for types. Instead a term of type `A : Ty` Γ is a *retraction* `t` $: \Gamma$`.A` $\to \Gamma$ of the inclusion ι $: \Gamma \to \Gamma$`.A`. The intuition is that if there is a term of type `A` in Γ, then `t` maps the variable in Γ`.A` to this term in Γ. Since `t` is a retraction (i.e. `t`$\circ\iota$ `= id` Γ) it leaves Γ fixed, so that the only information recorded in `t` is the term. As an example, the null vector may be a term `null : (n:`$\mathbb{N}$`,v:Vec n)` $\to$ `(n:`$\mathbb{N}$`)`.

[37] Hofmann 1997.

This completes our description of categories with attributes. Of course, while our intuition is taken from the syntactic category of (some unspecified) dependent type theory, any category with the mentioned structure will do.

A category with definitions will be much like a CwA, but with an explicit term functor which assigns each type in context to a set of available terms, and a new operation which extends a context with a *name for a term*. The idea is that when a context is extended with a name for a term, the term becomes a part of the context — or in other words it is defined — and can be used to make new types and terms. Retractions of inclusions are not identified with terms, but are given the role of representing names for terms already constructed. Thus, every retraction of an inclusion gives rise to a term, but not all terms will have an associated retraction.

3.2 Two simple examples

Before we give the definition of a category with definitions, we will consider two examples of CwAs where there are no complicated dependency structure on the types. Hopefully these can serve to give some intuition for the definition to come.

Example E:1. First we will consider the category `FinSet`, and the constant functor `Ty : FinSet` $\to$ `Set`, defined by `Ty` $\Gamma :=$ `1`, for every Γ `: FinSet`. The intuition is that we are in a single type setting, where our contexts are merely sets of variables — all of the same type — and we consider no other terms. Context extension is defined by Γ`.`$* :=$ Γ `+ 1` and σ`.`$*$ `=` σ `+ id 1`, and the inclusion $\iota :=$ `inl`.

The terms `t :` Γ `+ 1` $\to \Gamma$, namely retractions of ι, are just the elements of Γ — so all terms are variables.

Example E:2. For our next example we will try to motivate how to consider terms part of the context, and how variables and terms interact.

The idea behind this example is to have a single type, which we think of as natural numbers, with two term constructors, `zero` and `succ`, which we can use to extend the context with a defined element. For instance, `(x:N, y:N; zero; succ x; succ y; succ (succ x))` would be the context with two variables where we have constructed the terms `zero` `succ x`, `succ y` and `succ (succ x)`.

Let `bot : Poset` $\to$ `Poset` be the functor freely adjoining a bottom element, $\bot$, and let `F : Poset` $\to$ `Poset` be the functor `F X = bot (X`

`+ 1)`. We let η `: Id Poset` $\Rightarrow$ `bot` be the obvious inclusion of a poset `X` into `bot X`.

The objects of our category of contexts, $\mathbb{C}$, will be finite sets `X`, considered as discrete posets, equipped with functions `m : X` $\to$ `F X`. Morphism `(X,m)` $\to$ `(Y,n)` will be functions ϕ `: X` $\to$ `Y` such that for all `x:X` we have `(F` ϕ $\circ$ `m) x` $\leq$ `(n`$\circ\phi$`) x`.

Again, `Ty` $: \mathbb{C} \to$ `Set` will be a constant functor, `Ty` Γ `=` $\{$`N`$\}$, and our intuition this time is that our single type, `N`, is the type of natural numbers. A context Γ `= (X,m)` represents a set of "natural numbers" `X`, some of which are variables — namely those `x : X` for which `m x` $= \bot$ — and others are either

- successors: `m` mapping them to their predecessor in `(`$\eta\circ$`inl)(X)` $\subseteq$ `F X`
- or zeros, `m` mapping them to η `(inr` $*$`)`.

The restriction on the morphisms, `F` ϕ $\circ$ `m` $\leq$ `n` $\circ$ ϕ, ensures that variables can be instantiated, and zeros and successors must be preserved. If one is concerned about circular successor chains one can further add a well-foundedness requirement on the objects.

Context extension in for category is given by

$$\texttt{(X,m).N} := \texttt{(X+1,m+(const}\ \bot\texttt{))},$$

which adds a single element which is mapped to $\bot$ (i.e. which is a variable). We see again that the terms in each context, Γ `= (X,m)`, are exactly the elements of `X`. Since any context has only finitely many elements, we need to change context in order to access more elements of `N` — and this is where we introduce the notion of extending a context with a term.

3.3 Categories with definitions

Definition E:3. A *category with definitions* (CwD) consists of

1. A category $\mathbb{C}$ `: Cat`, called the *category of contexts*.
2. A functor `Ty` $: \mathbb{C} \to$ `Set`, called the *type functor*. We will write `A[f]` for `Ty f A`.
3. A functor `Tm` $: \sum$`(`$\mathbb{C}$`,Ty)` $\to$ `Set`, called the *term functor*. We will write `a[f]` for `Tm f a`.
4. A functor `-.-` $: \sum$`(`$\mathbb{C}$`,Ty)` $\to \mathbb{C}$, called the *context extension by variable functor*.

5. A functor `-;-` : $\sum(\sum(\mathbb{C},\mathtt{Ty}),\mathtt{Tm}) \to \mathbb{C}$, called the *context extension by definition functor*. We will write $\Delta;\mathtt{a}$ instead of the more verbose $(\Delta,\mathtt{A});\mathtt{a}$.
6. A natural transformation $\iota : \pi_0 \to \mathtt{-.-}$, whose components we will denote $\iota\ \mathtt{A} : \Delta \to \Delta.\mathtt{A}$, and such that the naturality squares for ι are push-outs.
7. A natural transformation $\chi : \mathtt{-.-} \circ \pi_0 \to \mathtt{-;-}$, whose components we will denote $\chi\ \mathtt{a} : \Delta.\mathtt{A} \to \Delta;\mathtt{a}$, and such that the naturality squares for χ are push-outs.
8. For each $\mathtt{A} : \mathtt{Ty}\ \Gamma$ a term $\mathtt{v\,A} : \mathtt{Tm}\ (\Gamma.\mathtt{A},\mathtt{A}[\iota])$, such that

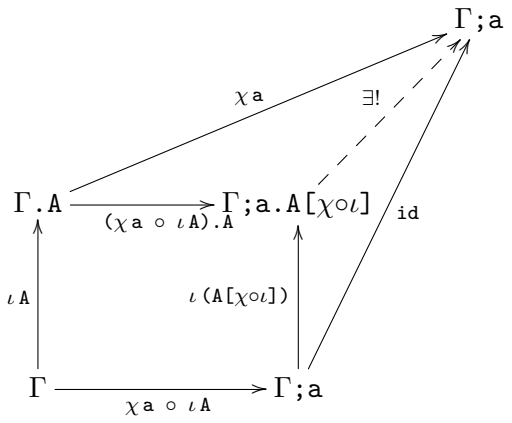
   a) $(\mathtt{v\,A})[\mathtt{f.A}] = \mathtt{v}\ (\mathtt{A}[\mathtt{f}])$
   b) $(\mathtt{v\,A})[\chi\,\mathtt{a}] = \mathtt{a}\,[\chi\,\mathtt{a} \circ \iota\,\mathtt{A}]$, for every term $\mathtt{a} : \mathtt{Tm}\ \Gamma\ \mathtt{A}$.
   c) $\chi\,(\mathtt{v}[\mathtt{f}]) \circ \iota\,(\mathtt{A}[\mathtt{f}]) \circ \mathtt{f} = \chi\,(\mathtt{v}[\mathtt{f}]) \circ (\mathtt{f} \circ \iota\,\mathtt{A}).\mathtt{A}$, for every $\mathtt{f} : \Gamma.\mathtt{A} \to \Delta$.[38]

We will refer to a category with definitions as a tuple $(\mathbb{C},\mathtt{Ty},\mathtt{Tm},\mathtt{-.-},\mathtt{-;-},\iota,\chi,\mathtt{v})$.

Remark E:4. The easiest way to compare categories with definitions to categories with attributes is to notice that conditions 1,2,4 and 6 are exactly the definition of a CwA.

If we only had conditions 1–7, then we could get a CwD from any CwA by simply letting `Tm` be the constant empty set functor. However, condition 8 ensures each retraction of an inclusion gives rise to a unique term. The context $\Gamma;\mathtt{a}$ always has a retraction $\Gamma;\mathtt{a.A}[\chi\circ\iota] \to \Gamma;\mathtt{a}$ representing $\mathtt{a}[\chi\circ\iota]$, given by the push-out property (see (1)), and is therefore referred to as the context extending Γ with a name for `a`.

[38]This rule ensures that sections map injectively into terms, by identifying maps which would discriminate between the section induced by `f` itself and the section induced by `v[f]`, in $\Delta;\mathtt{v}[\mathtt{f}]$. One could avoid this by replacing this rule with one that states that `Ty` and `Tm` maps the left hand side to to the same maps as the right hand side, without them always being equal. Or, in the even weaker case, that there are a coherent family of bijections $\mathtt{Tm}\ (\mathtt{B}[(\chi\,(\mathtt{v}[\mathtt{f}]) \circ \iota\,(\mathtt{A}[\mathtt{f}])) \circ \mathtt{f}]) \cong \mathtt{Tm}(\mathtt{B}[\chi\,(\mathtt{v}[\mathtt{f}]) \circ (\mathtt{f} \circ \iota\,\mathtt{A}).\mathtt{A})$ for every $\mathtt{B} : \mathtt{Ty}\ (\Gamma.\mathtt{A})$.



(1)

Example E:5. Continuing Example E:2, to make it a category with definitions, we define `Tm ((X,m),N)` := `(X + (X + 1))/`$\approx$ — the three summands representing already constructed terms, new successors, and new zeros, respectively — and $\approx$ is generated, co-inductively by:

- `inl x` $\approx$ `inr y` whenever η `y = m x`, where η `: Id` $\Rightarrow$ `bot` is the natural inclusion.
- `inl x` $\approx$ `inl y` whenever `m x = `η `x'` and `m y = `η `y'` and `x'` $\approx$ `y'`.

We then have to identify morphisms, which pointwise map to equivalent names. That is, given ϕ,ψ `: (X,m)` $\to$ `(Y,n)` we define $\phi \approx \psi$ to hold whenever `inl (`ϕ `x)` $\approx$ `inl (`ψ `x)` for every `x : X`.

Context extension by term definition is then given by the cases:

$$(\mathtt{X},\mathtt{m});[\mathtt{inl}\,\mathtt{x}] := (\mathtt{X}+1,\mathtt{m}+\mathtt{const}(\mathtt{m}\,\mathtt{x})) \tag{2}$$

$$(\mathtt{X},\mathtt{m});[\mathtt{inr}\,(\mathtt{inl}\,\mathtt{x})] := (\mathtt{X}+1,\mathtt{m}+\mathtt{const}\,((\eta\circ\mathtt{inl})\,\mathtt{x})) \tag{3}$$

$$(\mathtt{X},\mathtt{m});[\mathtt{inr}\,(\mathtt{inr}\,*)] := (\mathtt{X}+1,\mathtt{m}+\mathtt{const}\,((\eta\circ\mathtt{inr})\,*)) \tag{4}$$

which respects $\approx$. The natural transformation χ is the obvious inclusion, and the variable term `v N := [inl (inr *)]`.

We see that in this example all the well-founded contexts can be constructed from the empty one, by repeated extensions with variables and defined terms. Although the type `N` has only finitely many terms in any given context — in particular, it has only one term in the empty context, namely "zero" (η `(inr *)`) — we can construct all natural numbers (and all iterated successors of variables) by extending the context,

one successor at the time. This exemplifies a kind of local finitist version of type theory, where at any stage only finitely many types and terms exists, but as we make these definite by extending our context more appears, so that in aggregate any term is can be accessed.

3.3.1 Example

The usual CwA of sets, also gives rise to a CwD. The underlying category is the opposite category of sets[39]. Types in Γ are sets `A` equipped with functions $\mathtt{A} \to \Gamma$. Terms are sections of these functions. Transport of types along functions is by push-out (pull-back in `Set`). Context extension with a type (A,f) passes to the context `A`. Extension of a context with terms does nothing.

3.3.2 Syntax

Although we will not give a complete syntactic equivalent to that of a CwD, we will try to give some intuition as to what a syntactic counterpart would look like.

Let us consider the structural rule of substitution. While it is not always an explicit rule of dependent type theory it is usually a derivable rule. It is the rule for any judgement `J`, given as

$$\frac{\Gamma,\ \mathtt{x{:}A} \vdash \mathtt{J} \qquad \Gamma \vdash \mathtt{a} : \mathtt{A}}{\Gamma \vdash \mathtt{J[a/x]}}\ \mathtt{SUBST}$$

A syntactic counterpart to CwDs would abandon substitution in this sense, and rather use a special context $\Gamma\mathtt{;x{:=}a{:}A}$ which instantiates the variable `x` to the value `a`. The rules would look like the following:[40]

$$\frac{\Gamma \vdash \mathtt{a{:}A}}{\mathtt{(\Gamma;\ x{:=}a{:}A)\ context}}\ \mathtt{DEF,\ x\ fresh\ for}\ \Gamma$$

[39]The oppositeness can be rationalised by imagining that the context in `op Set` are actually all ways one could give values to the variables in the context. Thus the context is, morally, placed in the domain of a function — which is a "contravariant" position: the functor $\mathtt{op\ C} \times \mathtt{C} \to \mathtt{Set}$ mapping `(A,B)` to `Hom(A,B)` is contravariant in the `A` argument.

[40]We use the usual convention that if a judgement occurs in a rule its presuppositions are automatically assumed. For instance $\Gamma \vdash \mathtt{a{:}A}$ presupposes Γ `context` and $\Gamma \vdash \mathtt{A\ type}$.

$$\frac{\Gamma,\ \mathtt{x{:}A} \vdash \mathtt{J} \qquad \Gamma \vdash \mathtt{a{:}A}}{\Gamma;\ \mathtt{x{:=}a{:}A} \vdash \mathtt{J}}\ \mathtt{INST}$$

$$\frac{\Gamma,\ \mathtt{x{:}A} \vdash \mathtt{J} \qquad \Gamma \vdash \mathtt{a{:}A}}{\Gamma;\ \mathtt{x{:=}a{:}A} \vdash \mathtt{x} \equiv \mathtt{a} : \mathtt{A}}\ \mathtt{DEF\text{-}\equiv}$$

These rules are reflected in the semantic definition of categories with definitions. DEF is context extension by a definition, INST corresponds to the χ substitutions. Finally, DEF-$\equiv$ reflects the equality (v A)[χ a] = a [χ a $\circ$ ι A]. Note that $\equiv$ here denotes the usual judgemental / definitional equality of terms in the type theory.

Example E:6. To demonstrate how to use the above rules, consider a type theory with the following rules (in addition to the basic structural rules):

$$\frac{}{\Gamma \vdash \mathtt{N\ type}}\ \mathtt{N\text{-}FORM}$$

$$\frac{}{\Gamma \vdash \mathtt{0} : \mathtt{N}}\ \mathtt{0\text{-}I}$$

$$\frac{}{\Gamma,\mathtt{x{:}N} \vdash \mathtt{S\,x} : \mathtt{N}}\ \mathtt{S\text{-}I}$$

Then to construct the term SS0, we have the following derivations. For brevity we suppress the typing in the definitions.

$$\dfrac{\dfrac{}{\mathtt{x_1{:}N} \vdash \mathtt{S\,x_1{:}\ N}}\ \mathtt{S\text{-}I} \qquad \dfrac{\dfrac{\dfrac{}{\mathtt{x_0{:}N} \vdash \mathtt{S\,x_0{:}N}}\ \mathtt{S\text{-}I} \qquad \dfrac{}{\vdash \mathtt{0{:}N}}\ \mathtt{0\text{-}I}}{\mathtt{x_0{:=}0} \vdash \mathtt{S\,x_0{:}N}}\ \mathtt{INST}}{\mathtt{x_0{:=}0,\ x_1{:}N} \vdash \mathtt{S\,x_1{:}\ N}}\ \mathtt{INST}}{\mathtt{x_0{:=}0;x_1{:=}S\,x_0} \vdash \mathtt{S\,x_1\ :\ N}}$$

Notice how a type theory along these lines would be sensitive to the difference between the formulation of S-I above and the rule:

$$\frac{\Gamma \vdash \mathtt{n} : \mathtt{N}}{\Gamma \vdash \mathtt{Sn} : \mathtt{N}}\ \mathtt{S\text{-}I'}$$

The rule S-I' would give infinitely many terms of type N without needing further extensions of the context.

3.4 Comparison with categories with families

Categories with families (CwFs) differ from CwAs by not having the lifts with the push-out property, and instead having an operation which takes an f : Γ → Δ and a term a : Tm (Δ,A[f]) to a map ⟨f,a⟩ : Γ.A → Δ. In addition one introduces a special variable term v A : Tm (Γ.A,A[ι]). These two are then subject to the equations

⟨f,a⟩ ∘ ι A = f
v[⟨f,a⟩] = a
⟨ι A, v A⟩ = id (Γ.A)
⟨g∘f,a[g]⟩ = g∘⟨f,a⟩

CwDs have both the variable term and the push-out properties. The CwF operation of extending a substitution with a term can be emulated by defining ⟨f,a⟩ : Γ.A → Δ;a by ⟨f,a⟩ := χ a ∘ f.A. This satisfies the equations:

⟨f,a⟩ ∘ ι A = (χ a ∘ ι (A[f])) ∘ f
v[⟨f,a⟩] = a[χ a ∘ ι (A[f])]
⟨ι A, v A⟩ = χ (v A)
⟨g∘f,a[g]⟩ = (g;a)∘⟨f,a⟩

As noted, every CwD has a CwD substructure, and this substructure, through the well known equivalence of CwAs with CwDs, also gives rise a definition of extending a substitution, but this time with a retraction. This extension is defined for every f : Γ → Δ and every retraction r : Δ.A[f] → Δ as...

[f,r] : Γ.A → Δ
[f,r] := r∘f.A

We have the following relationship between the two (by 8c in Definition E:3):

⟨f,v[r]⟩ = χ (v[r]) ∘ ι (A[r]) ∘ [f,r]

3.4.1 Essentially categories with attributes

We can now define a CwA as a CwD where the extra term structure coincides with retractions, and is thus redundant.

Definition E:7. A *category with attributes* is a category with definitions,
(ℂ,Ty,Tm,-.-,-;-,ι,χ,v), such that for all Γ : Ob ℂ and A : Ty Γ

- Γ;a = Γ, for all a : Tm (Γ,A), and
- χ a is a retraction of ι A for all a : Tm (Γ,A).

Remark E:8. This is not the standard definition of a CwA. But one can see that it is basically equivalent by observing:

- that we can turn all the arrows around,
- each term a : Tm (Γ,A) gives a retraction χ a of ι A,
- every retraction t : Γ.A → Γ gives a term v A[t] in A[ι][t] = A[t∘ι] = A[id Γ] = A, and
- χ (v[t]) = t for every retraction, by applying 8c) of the definition of a CwD.

Since v[χ_a] = a by the definition of a CwA, and χ (v[t]) = t, we see that there is a bijection between retractions of ι A and terms Tm (Γ,A), so for CwAs Tm, -;- and v are all superfluous.

Definition E:9. A category with definitions is *essentially CwA* if χ a ∘ ι A is an isomorphism for every a : Tm (Γ,A).

Remark E:10. The difference between CwAs and essentially CwAs is that essentially CwA structure is preserved by equivalence, while the definition of CwA includes an equality of objects, which may not be preserved by an equivalence.

We will now construct a left adjoint to the inclusion of essentially CwAs into CwDs. The idea is consider contexts up to "term telescoping", i.e. Γ;a_0;a_1;a_2;⋯, so that a type or term will belong to Γ if it appears in any term telescope on Γ. Likewise, a morphism from Γ to Δ may map Γ to any term telescope on Δ.

3.5 Morphisms

In order to actually say what our adjunction is, we first need to define what morphisms of CwDs are. There are a few possible choices, but let us stick with the following.

Definition E:11. Given two CwDs,
`C = (ℂ,Ty,Tm,-.-,-;-,ι,χ,v)` and
`C' = (ℂ',Ty',Tm',-.-',-;-',ι',χ',v')`, we define a morphism from `C` to `C'` to consist of

- a functor `F : ℂ → ℂ'`,
- a natural transformation `ϵ : Ty ⇒ Ty' ∘ F`,
- a natural transformation `δ : Tm ⇒ Tm' ∘ (F,ϵ)`,
- a natural isomorphism `ϕ : -.-' ∘ (F,ϵ) ⇒ F ∘ -.-`, and
- a natural isomorphism `ψ : -;-' ∘ ((F,ϵ),δ) ⇒ (F,ϵ) ∘ -;-`,
- such that
  - `Fι = ϕ ∘ ι'(F,ϵ)`,
  - `(F,ϵ)χ = ψ ∘ χ'((F,ϵ),δ)`, and
  - `δ (v A) = v (ϵ t)` for every `Γ : Ob ℂ` and `A : Ty Γ`.

Remark E:12. Since the morphisms are functors, there is a natural concept of 2-morphism associated — meaning that we can form a 2-category of CwDs with natural transformations satisfying four natural conditions. However, we will not give any results concerning this 2-categorical nature of CwDs. We will only use the fact that the collection of morphism, `CwD(C,D)`, between two CwDs forms a category when stating two conjectures.

Definition E:13. Given two CwDs, `C = (ℂ,Ty,Tm,-.-,-;-,ι,χ,v)` and `C' = (ℂ',Ty',Tm',-.-',-;-',ι',χ',v')`, and two CwD morphisms `ℱ = (F,ϵ,δ,ϕ,ψ)` and `ℱ' = (F',ϵ',δ',ϕ',ψ')` we define a 2-morphism from $\mathcal{F}$ to $\mathcal{F}'$ to consist of

- a natural transformation `α : F → F'`, such that
- `Ty'α ∘ ϵ = ϵ'`,
- `Tm'(α') ∘ δ = δ'` where `α' : (F,ϵ) ⇒ (F',ϵ')` is given by `α'(x,A) := α x : (F x, ϵ x A) → (F' x, ϵ' x A)`,
- `α(-.-)∘ϕ = ϕ'∘(-.-)'α'`, and
- `α'(-;-)∘ψ = ψ'∘(-;-)'α''` where `α' : ((F,ϵ),δ) ⇒ ((F',ϵ'),δ')` is given by `α'' ((x,A),a) := α x : ((F x, ϵ x A),δ x A a) → ((F' x, ϵ' x A),δ' x A a)`,

3.6 Free essentially CwAs

We will now construct a free essentially CwA given any category with definition.

Definition E:14. Given a category with definitions `C = (ℂ,Ty,Tm,-.-,-;-,ι,χ,v)` and a context `Γ : Ob ℂ`, denote by `T Γ` the collection of sequences `t : Ob ∑(∑(ℂ,Ty),Tm)*` such that

- `(π₀∘π₀) (t 0) = Γ`, if the length of `t` is greater than `0`.
- `(π₀∘π₀) (t (i+1)) = (-;-) (t i)` for all `i+1 < len t`

We will call the elements of `T Γ` *term telescopes*.

We denote by `head t : Ob ℂ` the final extended context of `t`, namely `head t := (-;-) (t (len t - 1))`, if the length of `t` is greater than zero, and `head () = Γ`.

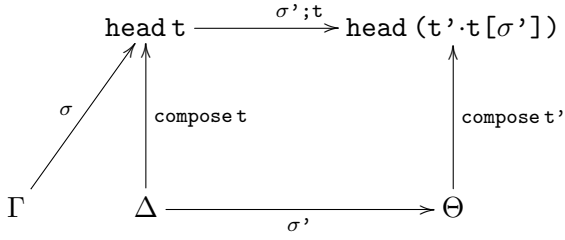
Definition E:15. Given a category with definitions `C = (ℂ,Ty,Tm,-.-,-;-,ι,χ,v)` and a context `Γ : Ob ℂ` and a term telescope `t : TΓ`, we define a functor `chain t : [(len t) + 1] → ℂ`, where `[(len t) + 1]` is the finite linear order of length `(len t) + 1` considered as a category, by

- `chain t 0 := Γ`,
- `chain t (i+1) := (-;-) (t i)` for all `i < len t`, and
- `chain t (r i) := (χ ((π₁∘t) i))∘ι ((π₁∘π₀∘t) i)`, on morphisms `r i : i → i+1`.

`compose (chain t) : Γ → head t` denotes the composition of all morphisms in the chain.

Definition E:16. Given a category with definitions `C = (ℂ,Ty,Tm,-.-,-;-,ι,χ,v)` we define the category `ℂ[(χ∘ι)⁻¹]` by

- `Ob ℂ[(χ∘ι)⁻¹] := Ob ℂ`
- `Mor(Γ,Δ)` is the collection of tuples `(t,σ)` such that
  - `t : TΔ`
  - `σ : Γ → head t`
  - where we identify `(t,σ)` and `(t',σ')` whenever there is a `u : TΔ` and morphisms `f : head t → head u` and `f' : head t' → head u` under `Δ`, such that 'f ∘ σ = f' ∘ σ'.
- Composition of `(t,σ) : Γ → Δ` and `(t',σ') : Δ → Θ` is given by transporting `t` along `σ'` and appending it to `t'` to obtain a longer term telescope, and then composing `σ` with the lifting of `σ'`. See diagram (5).



(5)

Definition E:17. Given a category with definitions
`C = (ℂ,Ty,Tm,-.-,-;-,ι,χ,v)` we define the category with definitions $\mathtt{C}^+$ by
$\mathtt{C}^+$:= `(ℂ⁺,Ty⁺,Tm⁺,-.-⁺,-;-⁺,ι⁺,χ⁺,v⁺)` where

- $\mathbb{C}^+ := \mathbb{C}[(\chi\circ\iota)^{-1}]$ as above,
- $\mathtt{Ty}^+\,\Gamma$:= { `(t,A) | t : T Γ ∧ A : Ty (head t)` } / ≈, where `(t,A)` ≈ `(t',A')` whenever there is a `u : TΔ` and morphisms `f : head t → head u` and `f' : head t' → head u` under Δ, such that `A[f] = A'[f']`,
- $\mathtt{Tm}^+\,(\mathtt{t},\mathtt{A})$:={`(u,a) | u : T (head t) ∧ a : Tm (head t, A[chain u]`}/≈, where `(u,a)` ≈ `(u',a')` whenever there is a `u'' : TΔ` and morphisms `f : head u → head u''` and `f' : head u' → head u''` under Δ, such that `a[f] = a'[f']`,
- `Γ.(t,A)` := `(head t).A`,
- `Γ;(u,a)` := `(head u);a`,
- ι`(v,A)` := `((),ι A ∘ (compose (chain t)))`,
- χ`(u,a)` := `((),χ a ∘ (compose (chain u)).A)`, and
- $\mathtt{v}^+$`(t,A)` := `((),v A)`

Lemma E:18. For all `C : CwD` we have that $\mathtt{C}^+$ is essentially CwA.

Proof: Given Γ and `(t,A) : Ty⁺ Γ` and `(u,a) : Tm⁺ (Γ,(t,A))`, we must construct an inverse to χ`(u,a)` ∘ ι`(t,A)`.

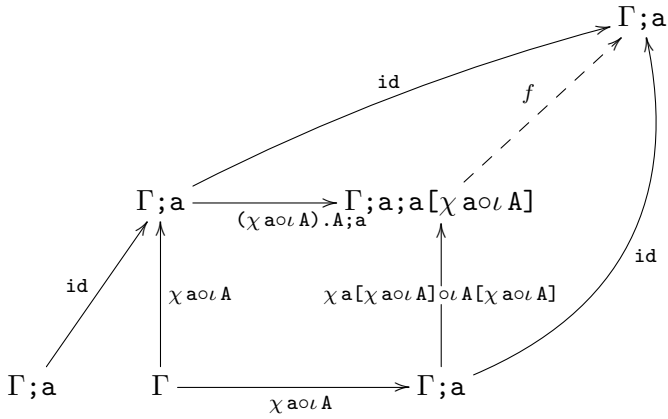
First observe the following simplification, where · denotes telescope composition.

```
  (χ (u,a) ∘ ι (t,A))
= ((), χ a ∘ (compose (chain u)).A ∘ ι A ∘ compose (chain t))
= ((), χ a ∘ ι (A[u]) ∘ compose (chain (t·u)))
```

The inverse can be given as,
σ := `(t·u·((head u,A[t]),a), id((head u);a))`. Composing from the left gives:

```
  σ ∘ (χ(u,a) ∘ ι(t,A))
= (v·u·((head u,A[t]),a), id((head u);a))
  ∘ ((), χa ∘ ι(A[u]) ∘ compose(chain(t·u)))
= (t·u·((head u,A),a), χa ∘ ι(A[u]) ∘ compose(chain(t·u)))
≈ ((),idΓ)
```

To show that composition form the right also yields the identity, one can spell out the composition and use the push-out properties of χ and ι. For illustration we will consider the special case `u = v = ()`, which still captures the general idea.



(6)

The square in diagram (6) is a push out, and `f` is the unique morphism given by the universal property of push-outs. Together with the identity, `f` shows that χ`((),a)` $\circ$ ι`((),A)` $\circ$ σ $\approx$ `id`$\{\Gamma;a\}$. □

Theorem E:19. The inclusions of essentially CwAs into CwDs has a left adjoint, namely $-^+$.

Proof: The unit of the adjunction is the map η : `C` $\to$ `C`$^+$ given by η`C`Γ := Γ and η`C`σ := (Δ,σ) (where $\sigma : \Gamma \to \Delta$). The actions on types and terms are similarly simple inclusions.

The co-unit is the map ϵ : `C`$^+$ $\to$ `C` which for every essentially CwA `C`, maps ϵ`C`Γ = Γ and ϵ`C(v,`σ`) = (compose (chain v))`$^{-1}\circ\sigma$. Types and terms are also transported along `(compose (chain v))`$^{-1}$.

The induced maps between $\mathtt{C}^+ \to \mathtt{D}$ and $\mathtt{C} \to \mathtt{D}$ (for $\mathtt{D}$ essentially CwA), form a bijection. □

Remark E:20. Theorem E:19 can be seen as a kind of definition elimination. It also demonstrates that if one is interested in models as CwD morphisms into `op Set`, which is a CwA, and hence essentially CwA, then the definition structure on the CwD is irrelevant. This matches the traditional expectations of definitions — namely that they can be eliminated and have no impact on semantics. However, we are free to consider semantics of a CwD as morphism into some CwD which is not essentially CwA, to get semantics which does not obey the common expectations.

3.7 Analogy with sharing

Sharing is a term used in computing when structurally equal parts of a data structure are stored at the same memory location. For instance, to store the expression `2·(2+2)·(2+2)`, one would not need to store the number `2` in five different memory locations, neither would one need to store the summation `(2+2)` twice. It suffices to store the terms once, making sure that they are correctly referenced elsewhere. This technique saves memory, at the cost of spending time figuring out which substructures can be shared.

In mathematics there is a clear analogue with definitions. We would rather not cite the term $\sum_{n=0}^{\infty} 1/n!$ every time, instead of using the well known constant e. Neither would we like to say "a set equipped with a binary operation, which is associative, has an identity and inverses" every time we meant "group". Choosing the correct definitions can be an art, and some times the most complicated expressions become clear and understandable when the subterms are given appropriate names. Giving names to phenomena, mathematical or otherwise, is key to recognising patterns.

Categories with definitions can be seen as an attempt to mimic the process of sharing in the semantics of dependent type theory. Just as structures are committed to memory, terms extend the contexts of a CwD. Once the context is extended, it can be used to form several new terms, without needing to reconstruct it every time.

4 One-way Categories

In this section we give a short introduction to one-way categories.

The basic idea is that each object in the category represents a type, and an arrow represents a dependency between types. Composition expresses the transitivity of dependency. Identities are more or less just along for the ride. Given such a type category, a functor to `Set` is a structure instantiating each type with a set, dependencies represented by functions.

Definition E:21. A outward one-way category is a category $\mathbb{C}$ such that the relation $\prec$ `: Ob` $\mathbb{C} \to$ `Ob` $\mathbb{C} \to$ `Prop`, defined by `x` $\prec$ `y` := $\exists$`f : Hom` $\mathbb{C}$ `x y  f` $\neq$ `(id x)`, is well founded, in the sense that for any `P : Ob` $\mathbb{C} \to$ `Prop` such that `(`$\forall$ `x  (`$\forall$ `y (x`$\prec$`y` $\to$ `P y)) `$\to$ `P x)` we have `P x` for all `x : Ob` $\mathbb{C}$.

Furthermore, we require that given any `f : Hom` $\mathbb{C}$ `x y` either `y = x` $\wedge$ `f = id x` or $\neg$`(x = y` $\wedge$ `f = id x)`.

Definition E:22. An *inward one-way category* is a category, $\mathbb{C}$ `: Cat`, such that `op` $\mathbb{C}$ is outward one-way.

Remark E:23. The *outward* and *inward* direction come from the direction of arrows, out from an object, and into an object. An outward one-way category has no infinite chain of composable, non-identity morphisms going out from any given object.

Remark E:24. Though one-way categories are categories, and we will consider functors defined on them – one-wayness is not really a property of categories, since the notion does not respect equivalence of categories. It is more appropriate to consider one-way categories as a kind of combinatorial object of their own.

Remark E:25. We will from here on drop the prefix "outward" when talking about outward one-way category, since outward will be our default one-wayness.

4.1 Structures

We will now define the notion of a structure for a one way category. The idea is that we need a set for each sort: say `A x` is the set of elements of

sort `x` in the structure `A`. Then, for each dependency we need a compatible function assigning each element of the structure to the parameters of its sort: say `f : x → y` is a dependency between the sorts `x` and `y`, then for each `a : A x` we need to assign some `A f a : A y`.

Definition E:26. Given an (outward) one-way category, $\mathbb{C}$, we define the *category of* $\mathbb{C}$*-structures* to be `[ℂ,Set]`, the category of functors from from $\mathbb{C}$ to `Set`. We may some times refer to $\mathbb{C}$-structures as *shapes*.

A structure `A : [ℂ,Set]` is *finite* if `∑(ℂ,A)` is finite. We denote the full subcategory category of *finite* $\mathbb{C}$*-structures* by $\{\mathbb{C}$`,Set`$\}$.

Remark E:27. Makkai 1995 requires that top-level fibres of structures are subsingletons. This is a reasonable requirement, but we will not include this as part of our structures.

Remark E:28. A $\mathbb{C}$-structure, `A`, is sometimes better seen as the category `∑(ℂ,A)` living over $\mathbb{C}$ by π_0 `: ∑(ℂ,A)` $\to \mathbb{C}$. We can see the objects of `∑(ℂ,A)` as typed points, and if the type of an object is dependent on other types, the required arguments are found by following the arrows out of the object.

4.2 Extending structures

We will now define a way to extend $\mathbb{C}$-structures with a single element, of a given sort. This lays the foundation for the next section, and the extension operation gives a convenient layer of abstraction.

Definition E:29. Given a one-way category, $\mathbb{C}$, we let `y : op ℂ →` `[ℂ,Set]` denote the contravariant Yoneda functor. Given `x : Ob ℂ` we will say that `y x` is *the shape* of `x`.

We then define the functor `∂ : op ℂ → [ℂ,Set]` by

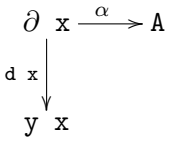
$$\partial\ \texttt{x y} = \texttt{Hom x y - id x} \text{ and}$$
$$\partial\ \texttt{f g} = \texttt{f} \circ \texttt{g}.$$

Notice how we here use that $\mathbb{C}$ is a one-way category. Given `x : Ob ℂ` we will say that `∂ x` is the *border shape* of `x`.

Furthermore, we define a natural transformation `d : ∂ ⇒ y` by the obvious inclusion `Hom x y - id x` $\subseteq$ `Hom x y`.

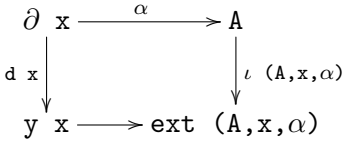
Definition E:30. We define the functor $\mathcal{E}$: [ℂ,Set] → Set by mapping a structure A : Ob [ℂ,Set] to the set of pairs, (x , α), where x : Ob ℂ and α : ∂ x ⇒ A. We define $\mathcal{E}$ on natural transformations, ϵ : A ⇒ B, by $\mathcal{E}$ ϵ (x, α) = (x, $\epsilon \circ \alpha$).

Definition E:31. We define an action of $\mathcal{E}$ on [ℂ,Set], namely ext : $\sum$([ℂ,Set],$\mathcal{E}$) → [ℂ,Set], by letting ext (A, x, α) be the push out of the diagram...



...in [ℂ,Set].

Definition E:32. We denote by ι : π_0 ⇒ ext the natural transformation given by the right hand side of the push out diagram:



Definition E:33. We denote by $*$: ext (A,x,α) x the image of id x : y x x by the push-out of α by d x (the lowermost arrow in the above diagram).

Remark E:34. One can regard ℂ-structures as a kind of cell complexes. Each object of ℂ represents a kind of face, and the maps out of an object describes the subfaces. There is a canonical ℂ-structure y x for each x : Ob ℂ given by the Yoneda embedding. The ext construction takes as input the border of a face α : ∂x ⇒ A and glues a face to it, to obtain ext (A,x,α). The new face of kind x lies in ext (A,x,α) x and it is this face we denote by $*$.

Example E:35. The category Δ_+, of finite, inhabited, linear orders and injective monotone maps is an inwards one-way category. Thus op Δ_+ is a one-way category. The category [op Δ_+,Set] is of course the

well known category of semisimplicial complexes. Given a semisimplicial complex `A : Ob [op Δ+,Set]`, the set $\mathcal{E}$ `A` is the set of ways we that we could possibly glue a new face to `A`. Given a face `f : ` $\mathcal{E}$ `A` then `ext (A,f)` is the complex resulting from gluing `f` to `A`. Notice that `f : ` $\mathcal{E}$ `A` has two components; `f` $\equiv$ `([n+1],`α`)`, where `[n+1]` is an `n`-simplex[41], and α `:` ∂ `[n+1]` $\Rightarrow$ `A` which maps the border of the simplex into A. The border of `[n+1]`, namely ∂ `[n+1]`, coincides with the usual notion of border of a simplex.

4.3 The category with attributes of structures

Proposition E:36. For any one-way category $\mathbb{C}$, we can form a category with attributes structure on `[`$\mathbb{C}$`,Set]` such that `Ty` := $\mathcal{E}$ and `-.-` := `ext`.

Proof: The required push-out properties are satisfied by definition of $\mathcal{E}$. Terms are simply the sections.

Remark E:37. The above proposition is the formalisation of the intuition stated previously about how a one-way category forms a type system. We conjecture that if we define a new kind of semantics $\mathbb{C}$ by considering morphisms into other CwDs, this specialises to the semantics we already know, in the case of the `op Set`.

Conjecture E:38. `op [`$\mathbb{C}$`,Set]` $\cong$ `CwA([`$\mathbb{C}$`,Set], op Set)`.

Propositions E:39. The inclusions $\Gamma \to \Gamma$`.A` in the CwA structure on `[`$\mathbb{C}$`,Set]` are monomorphisms.

Proof: By definition ι `A : `$\Gamma \to \Gamma$`.A` is a push-out of a mono. In `[`$\mathbb{C}$`,Set]` any push-out of a mono is mono.

Remark E:40. A CwA in which the inclusions are monomorphisms (or, in more standard/opposite terms the projections Γ`.A` $\to \Gamma$ are epi) can in some way be seen as being *intensional*. From a type theoretical perspective this means that if $\Gamma \vdash$ `a,a' : A` and Γ`.(x:B)` $\vdash$ `a` $\equiv$ `a' : A` then already $\Gamma \vdash$ `a` $\equiv$ `a' : A`. This principle is called *strengthening*. This is not true in extensional models such at the usual CwA structure on Set; consider extending a context with a variable of type $\emptyset$. Type theory with equality reflection, sometimes called extensional

[41] We use the notation `[k]` to denote the canonical linear order with `k` elements, not `k+1` elements

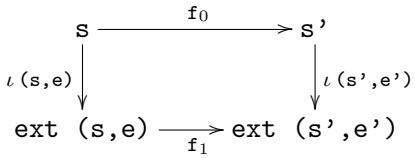
type theory, also fails this property, since `Id A a a' ⊢ a ≡ a' : A` for any `a, a' : A`. But it *is* true in the syntactic CwA of intensional type theory.[42]

4.4 The category of corollas

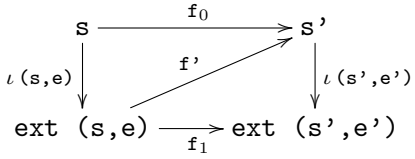
Now we will define a category of corollas, which will form the basis of the next session.

Definition E:41. Given a one-way category $\mathbb{C}$, *the category of* $\mathbb{C}$*-corollas*, $\nabla\ \mathbb{C}$, is defined by:

- `Ob (∇ ℂ) = {(s,e) | s : Ob [ℂ,Set] , e : ℰ s}`
- `Mor (s,e) (s',e')` is the set consisting of pairs `(f₀,f₁)` such that
  - `f₀ : s → s'`
  - `f₁ : ext s e → ext s' e'`
  - `(ι (s',e')) ∘ f₀ = f₁ ∘ (ι (s,e))` (see diagram below).



Definition E:42. Given a morphism `f ≡ (f₀,f₁) : (s,e)→(s',e')` we say that `f lifts` if there is `f* : ext (s,e) → s'` such that `f₀ = f* ∘ ι (s,e)` and `f₁ = ι (s',e') ∘ f*`.



Lemma E:43. For each `f` such that `f lifts` the lift is unique.

Proof: `ι (s',e')` is a monomorphism.

[42] For an example of strengthening for a dependent type system, see Theorem 6.6 p. 35 of Harper and Pfenning 2005.

Lemma E:44. For any morphism $(f_0,f_1) : (s,e) \to (s',e')$ in $\nabla\,\mathbb{C}$, if $f_1 * \neq *$ then f_1 lifts to a morphism $f* : ext\ s\ e \to s'$, such that $f* \circ (\iota\ (s,e)) = f_0$.

Proof: For each $x : \mathbb{C}$ we have that $ext\ s\ e\ x$ is a either just the image of $\iota\ (s,e)\ x$ or the disjoint union of $\iota\ (s,e)\ x$ and $\{*\}$ (this follows from the decidability requirement we put on one-way categories). Since $\iota\ (s,e)\ x$ is injective, we may map any element in it to an element of $s'\ x$ by seeing what f_0 does to the corresponding element in $s\ x$. For $*$ we know that $f_1\ *$ is in the image of $\iota\ (s',e')$, since it is not $*$, thus we map it to the corresponding element in s'.

Example E:45. The category of $(op\ \Delta_+)$-corollas is the category whose objects are semisimplicial complexes with a selected, top dimension face. The morphisms are maps of semisimplicial sets which map the unselected faces to unselected faces — possibly mapping the selected one to the selected one, but possibly mapping it to something else.

Definition E:46. Given a corolla $(s,e) : Ob\ (\nabla\,\mathbb{C})$ and a $\mathbb{C}$-structure A, we define *a pure, (s,e)-shaped* operation on A to be a function $\phi : (s \Rightarrow A) \to (ext\ (s,\ e) \Rightarrow A)$ such that $(\phi\ \alpha) \circ (\iota\ (s,\ e)) = \alpha$, for every $\alpha : (s \Rightarrow A)$.

Remark E:47. The term *"pure"* in this context refers to that the domain of the function is the entire set of natural transformations $(s \Rightarrow A)$. We regard s as a context, in the sense that we defined a CwA structure on $[\mathbb{C},Set]$. A more general notion of context will have some elements of s decorated with definitions. This will allow restricting the domain of the function, to give certain partial operations.

Example E:48. As an example of why such partial operations may be interesting, consider the well known horn-filling operation on (semi-)simplicial complexes. An $(n+1)$ dimensional horn is a subsimplex $H \subset \partial\ [n+2]$ obtained by removing a single (n-dimensional) face of $\partial\ [n+2]$ $[n+1]$. Given a $(n+1)$-dimensional horn H, a horn-filling operation on a simplicial complex A is an operation extending any $\alpha : H \Rightarrow A$ to a full $n+2$ simplex $y\ [n+2] \Rightarrow A$.

A horn-filling operation extends the A-horn with two new faces in A, so while it would make sense to define operations with multi-face outputs, we may also attempt see the horn filling as two separate operations — one which fills out the missing face of the boundary, and second one

which fills out the resulting boundary with a top face making a complete simplex. However, here one must be careful. Not every boundary of a simplex can be filled out – horn-filling only fills boundaries where one face is constructed by the first step of the horn-filling operation. If every boundary could be filled, the complex would have trivial homotopy. Thus, the last step, the boundary filling, is a partial operation, taking only boundaries where one face is constructed from the first step as input.

One way to describe the situation is to say that (for each horn `H`) we have two operations called ϕ_0 and ϕ_1, and they are arranged in a category, $\mathcal{R}$, with a single non-identity arrow, `f` : $\phi_1 \to \phi_0$. To each operation we associate a complex with an extension: ϕ_0 is assigned the horn `H` together with an extension $\mathtt{e}_0$: $\mathcal{E}$ `H` such that `ext (H,`$\mathtt{e}_0$`)` $\cong \partial$ `[n+2]`, while ϕ_1 is assigned ∂ `[n+2]` and an extension $\mathtt{e}_1$: $\mathcal{E}$ `(`∂ `[n+2])` such that `ext (`∂ `[n+2], `$\mathtt{e}_1$`)` $\cong$ `y [n+2]`. There is then an obvious pair of inclusion maps making this a functor `D : op` $\mathcal{R} \to$ ∇ `(op` Δ_+`)`.

5 One-way Term Systems

In this section we give a definition of a one-way term system and give basic properties of these.

The idea is to represent a rule introducing a $\mathbb{C}$-term as an extended $\mathbb{C}$-shape. Some parts of the shape of a rule are variables, another are already introduced terms, constructed by other rules. All the rules fit together in an one-way category, and in total the system of rules will be a functor on this category.

Definition E:49. Given a one-way category $\mathbb{C}$, we define a one-way term system on $\mathbb{C}$ consists of a pair `(`$\mathcal{R}$`,D)` where

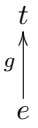
- $\mathcal{R}$ is a one-way category , and
- `D : op` $\mathcal{R} \to \nabla$ $\mathbb{C}$ is a functor,
- such that for each `x : Ob` $\mathcal{R}$ and each pair of morphisms `f :` $\mathtt{y}_0$ $\to$ `x`, `g :` $\mathtt{y}_1$ $\to$ `x`
  - if π_0 `(D` $\mathtt{y}_0$`)` = π_0 `(D` $\mathtt{y}_1$`)` and `(D f)`$_1$ `(D` $\mathtt{y}_0$`)` $*$ `= (D f)`$_1$ `(D` $\mathtt{y}_0$`)` $*$, then $\mathtt{y}_0$ `=` $\mathtt{y}_1$ and `f = g`.

We refer to the objects of $\mathcal{R}$ as *term-forming rules*, and we say that `D` assigns a rule to *its shape*.

While we have not yet defined terms, we will refer to, given `r : Ob` $\mathcal{R}$ and `x : Ob` $\mathbb{C}$, elements `p : ext (D r) x` as *subterms of sort* x occurring in the rule `r`. If `p` is the image of $*$ by $(\texttt{D f})_0$ `(D y)` for any `f : y` $\to$ `x` it is a *defined subterm*, else it is a *variable*.

Example E:50. A very simple example of a one-way category is the category with two objects, say `E` and `T`, a single non-identity arrow `typeOf : E` $\to$ `T`. Let us name this category `U`, for the time being. It can serve as a simple model of typed systems. `U`-structures are pairs of sets — one which we can think of as the set of types, and the other the set of expressions — along with a single function which assigns to each expression a type.

Say we wanted to introduce a type constant `t` and an expression of that type, say `e`. Then we would express that as a rule system, where $\mathcal{R}$ is the category...[43]



and where `D : op` $\mathcal{R}$ $\to$ ∇ `U` is the functor...

- sending `t` to `(`∂ `T,T, id (`∂ `T))`,
- sending `e` to `(`∂ `E, E, id (`∂ `E))`, and
- sending `g` to `(e,d E)` where `e :` ∂ `T` $\to$ ∂ `E` is the empty transformation (∂ `T` is the constant $\emptyset$ functor), and `d E : ext (D t)` $\to$ `ext (D e)` type-checks since `ext (D t) = ext (`∂ `T,T, id (`∂ `T)) = y T =` ∂ `E` and `ext (D e) = ext (`∂ `E,E, id (`∂ `E)) = y E`.

We see in this example that the rule `t` has a single subterm of sort `T`, namely the conclusion $*$ `: ext (D r) T` defined by the identity `id T`. On the other hand, `e` has two subterms: the conclusion and the subterm defined by `g`, namely $(\texttt{D f})_1$ `* : ext (D e) T`, of sort `T`.

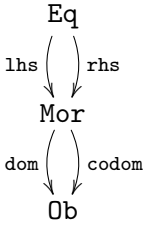
Example E:51. While rules of a term system cannot introduce equations, one can add a sort of "equalities" and consider the terms there

[43] $\mathcal{R}$ is only incidentally isomorphic to `U`, here.

as proofs of equality. As an example we formulate the rules for an E-category — that is a category, where the morphisms forms setoids rather than sets.

The underlying one-way category for these structure is the category, `E`, generated by the graph (7), and subject to the equations:

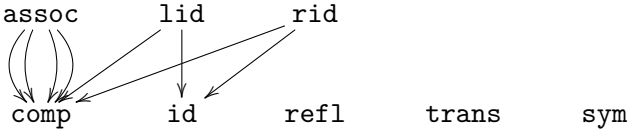
```
  dom ∘ lhs =   dom ∘ rhs
codom ∘ lhs = codom ∘ rhs
```



(7)

The structures in `[E,Set]` consist of a set of objects, a set of morphisms and a set of equations of morphisms. Each morphism has a domain object and a codomain object and each equation has a left hand side morphism and a right hand side morphism, such that the domain and codomain of each side agrees.

The category $\mathcal{R}$ of rules for an E-category is the category generated by the following graph:

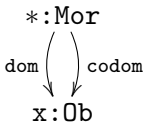


Each arrow in $\mathcal{R}$ represents an application of the rule in the codomain in the formulation of the domain. For example, the rule for associativity, `f∘(g∘h) = (f∘g)∘h`, contains four applications of the composition rule. Hence there are four arrows from `assoc` to `comp` — these will appear as the four nodes $\mathtt{c}_0, \cdots, \mathtt{c}_3$ in the diagram for `D assoc` (diagram 9). The shapes of each rule, given by $\mathtt{D} : \mathtt{op}\ \mathcal{R} \to \nabla\mathtt{E}$, may be illustrated by the following graphs, representing corollas in $\nabla\mathtt{E}$.

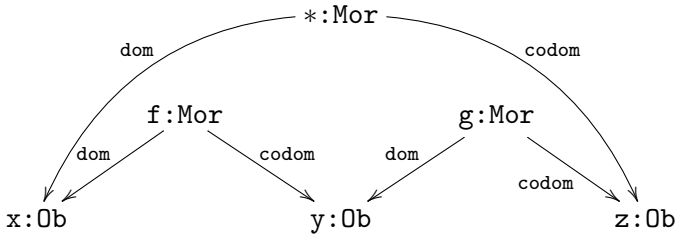
The nodes of the diagrams below are marked with sorts. For example the node `x:Ob` appearing in the diagram of `D id` means $\mathtt{x} \in \pi_0\,(\mathtt{D\ id})(\mathtt{Ob})$. Solid arrows show how arrows in $\mathcal{R}$ acts on the fibres.

For example the arrow appearing as `dom : f:Mor → x:Ob` in the diagram for `D(comp)` illustrates that π_0 `(D comp) dom f = x` — that is, the domain of `f` is `x`. Dashed arrows represents arguments for nodes defined by applying some rule. Finally, one node of each diagram is marked $*$, and represents the extension of the corolla.

`D id` is the shape in `[E,Set]` where the only non empty fibre is `E Ob` = {`x`}, along with an extension of a single morphism, marked $*$.

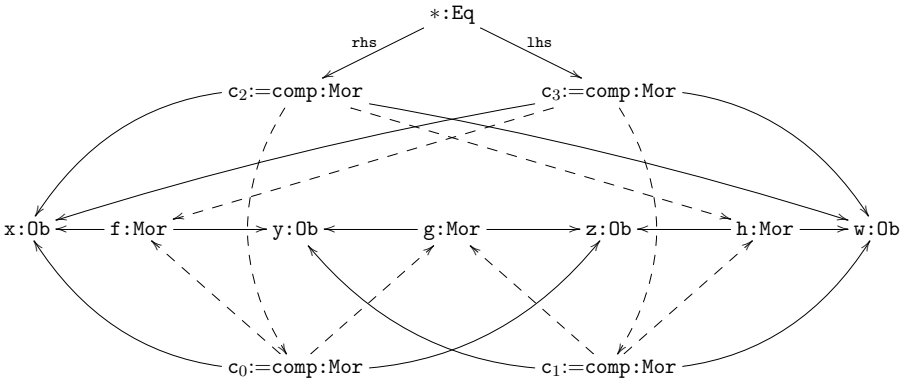


`D comp`...



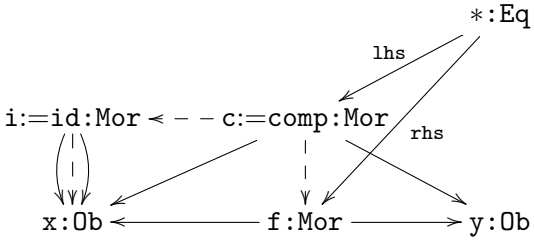
(8)

`D assoc` is illustrated in diagram 9. It introduces a new element in `Eq`. There are four arrows `assoc` → `comp` in $\mathcal{R}$ which means that we need to map the shape of `D(comp)` four times into the shape `D(assoc)`. These embeddings are illustrated with dashed lines – each composite arrow has a dashed arrow to its two components. For the sake of readability we have omitted the `dom` and `codom` decorations — for each arrow the domain is specified by the leftmost, solid arrow, and codom by the rightmost.

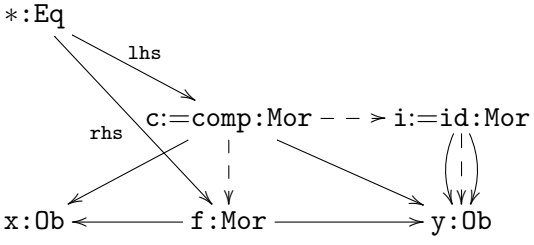


(9)

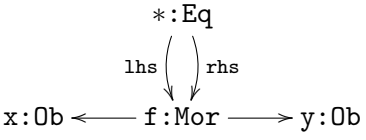
D lid...



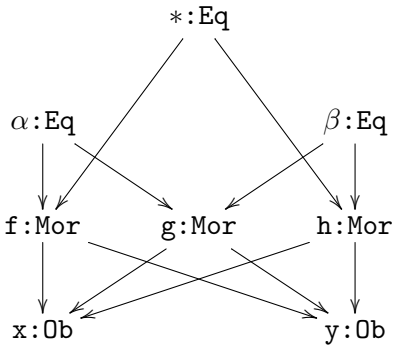
D rid...



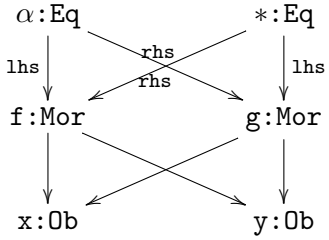
D refl...



D trans...



D sym...



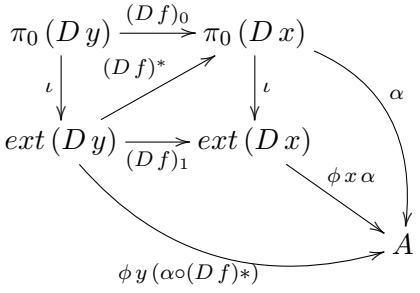
5.1 Structures

Definition E:52. Given a $\mathbb{C}$-structure A, a shape s : Ob [$\mathbb{C}$,Set] and extension e : $\mathcal{E}$ s, we define an *assignment* on A with shape (s,e), as a function ϕ defined on some subset dom $\phi \subseteq$ (s $\Rightarrow$ A), which takes every $\alpha \in$ dom ϕ to some $\phi\ \alpha$: ext s e $\Rightarrow$ A such that $\phi\ \alpha \circ (\iota$ (s,e)) = α.

Definition E:53. Given a $\mathbb{C}$-structure A, and a one-way term system ($\mathcal{R}$,D) on $\mathbb{C}$, an ($\mathcal{R}$,D)-structure on A consists of:

- an assignment ϕ x on A with shape D x, for each x : Ob $\mathcal{R}$,
- such that for each x : $\mathcal{R}$, we have that
  - dom (ϕ x) = { α : π_0 (D x) $\Rightarrow$ A | $\forall$ y $\forall$ (f : x$\rightarrow$y) (D f lifts) ($\alpha\circ$(D f)$_0$ $\in$ dom (ϕ y)) $\rightarrow$ ($\alpha \circ$ (D f)*)= (ϕ y ($\alpha\circ$(D f)$_0$)} (See diagram 10)

– for each `f : y → x` and α `: π₀ (D x) ⇒ A` we have that $(\phi\,\mathtt{x}\,\alpha)\circ(\mathtt{D\,f})_1 = (\phi\,\mathtt{y}\,(\alpha\circ(\mathtt{D\,f})_0)$ (See diagram 10)

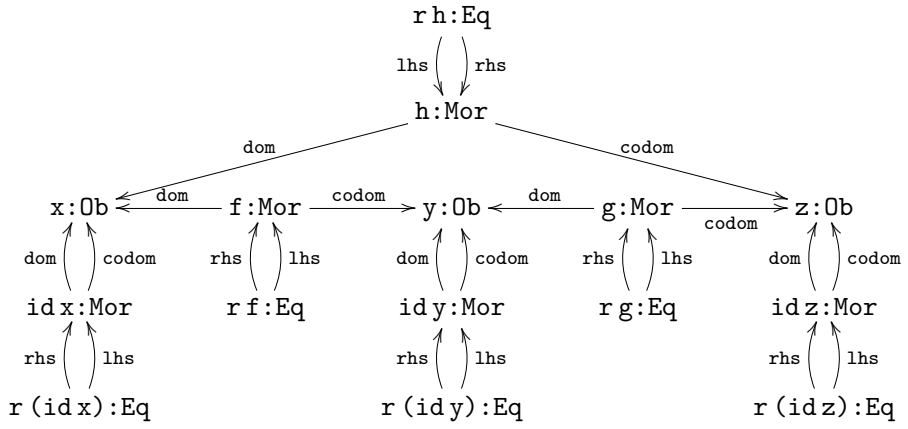


(10)

Example E:54. Let us illustrate how assignments work by instantiating Example E:51. An example `E`-structure `A` can be given by

- `A Ob` := {`x,y,z`},
- `A Mor` := {`f,g,h,id x,id y,id z`},
- `A Eq` := {`r f, r g, r h, r (id x), r (id y), r (id z)`}, where
- `A dom f` := `A dom (id x)` := `A codom (id x)` := `x`
- `A codom f` := `A dom (id y)` := `A codom (id y)` := `y`
- `A dom h` := `x`
- `A codom h` := `A dom (id z)` := `A codom (id z)` := `z`,
- `A dom h` := `y`,
- `A codom h` := `z`,
- `A lhs (r f)` := `A rhs (r f)` := `f`,
- `A lhs (r g)` := `A rhs (r g)` := `g`, and
- `A lhs (r h)` := `A rhs (r h)` := `h`,

or more succinctly, in diagram form...



(11)

There are ten possible maps[44] from π_0 (D comp) (diagram 8, p. 163) into our structure (11), that means that to specify an ($\mathcal{R}$,D) structure on A, we need to give a value to ϕ comp for each of those ten maps. In this case we do not have any choice, for each mapping π_0 (D comp) $\Rightarrow$ A there is a unique element of A Mor with the right domain and codomain which we can extend with to obtain a mapping ext (D comp) $\Rightarrow$ A.

Similarly for id, refl, trans, sym, assoc, rid and lid, in each case there are a number of mappings which fit, but for each we have a unique extension.

Example E:55. Going back to our motivating example of horn filling operations in Δ_+, we will define the following term system, ($\mathcal{R}$,D).

$\mathcal{R}$ is the category generated by the graph, which is the disjoint union over every horn H of dimension n+1:

$$\begin{array}{c} \phi_1\, H \\ \big\downarrow r\, H \\ \phi_0\, H \end{array}$$

D : op $\mathcal{R}$ $\to$ ∇(op Δ_+) is the functor mapping

- D (ϕ_0 H) := (H,$\mathtt{e}_0$) where $\mathtt{e}_0$ is such that ext (H,$\mathtt{e}_0$) $\cong$ ∂[n+2],

[44]Two identity compositions for each of the three non-identity morphisms, plus each identity morphism composed with itself, plus the composite g$\circ$f.

- `D (`ϕ_1 `H) :=  (ext (D (`ϕ_0 `H)),`$\mathtt{e}_1$`)`, where $\mathtt{e}_1$ is such that `ext (ext (D (`ϕ_0 `H)),`$\mathtt{e}_1$`) ≅ y [n+1]`, and
- `D (r H) :=` ι.

5.2 Structure homomorphisms

There is a natural notion of structure homomorphisms, for `(`$\mathcal{R}$`,D)` structures, forming a category `Struct(`$\mathcal{R}$`,D)`.

Definition E:56. Given two `(`$\mathcal{R}$`,D)` structures, `(A,`ϕ`)` and `(B,`ψ`)` we define an `(`$\mathcal{R}$`,D)`-homomorphism from `(A,`ϕ`)` to `(B,`ψ`)`, to consist of

- a natural transformation `f : A ⇒ B` such that
- ψ `r (f∘`α`) =` ϕ `r` α for each `r : Ob` $\mathcal{R}$, and α `:` π_0 `(D r) ⇒ A`.

Remark E:57. Definition E:56 is a quite strict definition of homomorphism. For instance, if we define E-category as in Example E:51, a homomorphism between E-categories is not quite the same as an E-functor since associativity proofs will have to commute on the nose. If one wants a more convincing model of categories, one can consider the subcategory where there are only reflexivity elements in `Eq`, as in Example E:54. This full subcategory of `Struct(`$\mathcal{R}$`,D)` will be equivalent to the category of small categories and functors.

5.3 Connecting term systems with CwDs

A term system `(`$\mathcal{R}$`,D)` is an object of quite concrete combinatorial nature. In the examples we have considered $\mathcal{R}$ and `D` have been finite. We would like to connect these finite objects with the usual categorical semantics of type theory, namely CwAs. In this final section we will give a conjecture up-front, and outline an attempt towards proving this conjecture.

Conjecture E:58. Given a term system `(`$\mathcal{R}$`,D)` we conjecture that there is a CwA `C(`$\mathcal{R}$`,D)` such that `op Struct(`$\mathcal{R}$`,D)` $\simeq$ `CwA(C(`$\mathcal{R}$`,D),op Set)`.

Conjecture E:59. Furthermore, we expect that in the case when $\mathcal{R}$ is finite and the corollas `D x` are finite for all `x : Ob` $\mathcal{R}$, that there is a locally finite CwD `CFin(`$\mathcal{R}$`,D)` such that
`op Struct(`$\mathcal{R}$`,D)` $\simeq$ `CwD(CFin(`$\mathcal{R}$`,D),op Set)`

5.3.1 Environments

While we do not yet have a proof of these conjectures, we will provide a few steps in the direction where we think a proof could be found. The main idea is to construct a CwD of $(\mathcal{R},\mathtt{D})$-environments which we then can complete to a CwD using the $-^+$ construction. As a first step we define a category environments, which could be taken as a starting point for a category with definitions.

The intuition behind the following definition is that an environment is a context in which some elements are given a definition as one of the rules in $\mathcal{R}$ applied to other elements of the environment. These definitions are structured as a functor $[\mathcal{R},\mathtt{Set}]$ which ensures correct applications of the rules according to their dependency structure. A co-cone collects all the defined terms and variables into a single structure.

Definition E:60. Given a one-way category $\mathbb{C}$ and a one-way term system on $\mathbb{C}$, $(\mathcal{R},\mathtt{D})$, we define an *$(\mathcal{R},D)$-environment* to consist of

- a $\mathcal{R}$-structure $\mathtt{s} : \mathcal{R} \to \mathtt{Set}$[45]
- a co-cone $(\mathtt{c},\epsilon)$ of $\mathtt{ext} \circ \mathtt{D} \circ \pi_0$, i.e. $\mathtt{c} : [\mathbb{C},\mathtt{Set}]$ and $\epsilon : \mathtt{ext} \circ \mathtt{D} \circ \mathtt{op}\ \pi_0 \Rightarrow \mathtt{const}\ \mathtt{c}$, where $\mathtt{op}\ \pi_0 : \mathtt{op} \sum(\mathcal{R},\mathtt{s}) \to \mathtt{op}\ \mathcal{R}$ is the obvious projection functor.
- such that:
  - If $\mathtt{r},\mathtt{r'} : \mathtt{Ob} \sum(\mathcal{R},\mathtt{s})$ are such that $\epsilon\ \mathtt{r} * = \epsilon\ \mathtt{r'} *$ in $\mathtt{c}$, then $\mathtt{r} = \mathtt{r'}$.

Definition E:61. A *morphism of $(\mathcal{R},D)$-environments* between two environments, $(\mathtt{s},(\mathtt{c},\epsilon))$ and $(\mathtt{s'},(\mathtt{c'},\epsilon'))$, consists of

- a map $\rho : \mathtt{s} \Rightarrow \mathtt{s'}$ of $\mathcal{R}$-structures,
- a map $\phi : \mathtt{c} \Rightarrow \mathtt{c'}$ of $\mathbb{C}$-structures,
- such that for each $\mathtt{r} : \mathtt{Ob} \sum(\mathcal{R},\mathtt{s})$ we have that $\phi \circ (\epsilon\ \mathtt{r}) = \epsilon'\ (\rho\ \mathtt{r})$

We denote by $\mathtt{E}_0 : \mathtt{Cat}$ the *category of $(\mathcal{R},D)$-environments* and morphisms between them.

[45] Note that $\mathcal{R}$-structure here does not refer to $(\mathcal{R},\mathtt{D})$-structure, but rather the notion defined in E:26 on page 155.

Definition E:62. We define $\mathtt{Ty_0 : E_0 \to Set}$ by $\mathtt{Ty_0} := \mathcal{E} \circ \mathtt{F}$ where $\pi_0 : \mathtt{F} \to \mathtt{[\mathbb{C},Set]}$ is the forgetful functor $\mathtt{F\,(s,\ (c,}\epsilon\mathtt{))} := \mathtt{c}$.

We define $\mathtt{Tm_0} : \sum\mathtt{(E_0,Ty_0)} \to \mathtt{Set}$ on objects $\mathtt{((s,\ (c,}\epsilon\mathtt{)),t)}$: $\mathtt{Ob}\ \sum\mathtt{(E_0,Ty_0)}$ by defining $\mathtt{Tm_0\,(\Gamma,t)}$, where $\Gamma = \mathtt{(s,\ (c,}\epsilon\mathtt{))}$ and $\mathtt{t} = \mathtt{(x,}\alpha\mathtt{)}$ to be disjoint union of

- $\{\beta : \mathtt{y\ x} \Rightarrow \mathtt{c} \mid \beta \circ \iota\,\mathtt{t} = \alpha : \partial\mathtt{x} \Rightarrow \mathtt{s}\}$, and
- the set of all pairs $\mathtt{(e,}\delta\mathtt{)}$ where
  - $\mathtt{e} : \mathcal{E}\ \mathtt{s}$, and
  - $\delta : \mathtt{ext} \circ \mathtt{D} \circ \mathtt{op}\ \pi_0 \Rightarrow \mathtt{const\ (ext\ (c,t))}$, is such that
    - $\delta\mathtt{(id}\ \mathcal{R},\ \iota\,\mathtt{e)} = \iota\,\mathtt{t} \circ \epsilon$, where $\mathtt{(id}\ \mathcal{R},\ \iota\mathtt{)} : \sum\mathtt{(}\mathcal{R}\mathtt{,s)} \to \sum\mathtt{(}\mathcal{R}\mathtt{,ext\ (s,e))}$ is the obvious inclusion.
    - $\delta\,\mathtt{(}\pi_0\,\mathtt{e,}*\mathtt{)}\,* = *$.

Finally, we define:

- $\mathtt{(s,(c,}\epsilon\mathtt{)).t} := \mathtt{(s,(ext\,(c,t),}\iota\,\mathtt{t}\circ\epsilon\mathtt{))}$
- $\mathtt{(}\rho\mathtt{,}\phi\mathtt{).t} := \mathtt{(}\rho\mathtt{,\ ext\,t\,}\phi\mathtt{)}$
- $\Gamma\mathtt{;(inr\,}\beta\mathtt{)} := \Gamma$
- $\mathtt{(s,(c,}\epsilon\mathtt{));(inl\,(e,}\delta\mathtt{))} := \mathtt{(ext\,(s,e),(ext\,(c,t),}\delta\mathtt{)}$, where $\mathtt{(e,}\delta\mathtt{):Tm\,t}$
- $\mathtt{(}\rho\mathtt{,}\phi\mathtt{);(inr\,}\beta\mathtt{)} := \mathtt{(}\rho\mathtt{,}\phi\mathtt{)}$
- $\mathtt{(}\rho\mathtt{,}\phi\mathtt{);(inl\,(e,}\delta\mathtt{))} := \mathtt{(ext\,}\rho\mathtt{,}\phi\mathtt{)}$
- $\iota_0\,\mathtt{t} := \mathtt{(id,}\iota\,\mathtt{t)}$ (by a slight overloading of notation).
- $\chi_0\,\mathtt{(e,}\delta\mathtt{)} := \mathtt{(}\iota\,\mathtt{e,}\iota\,\mathtt{t)}$, and
- $\mathtt{v_0\,t} := \mathtt{inr\,(}\pi_0\,\mathtt{t,}*\mathtt{)}$.

Remark E:63. The structure $\mathtt{(E_0,Ty_0,Tm_0,\text{-}.\text{-},\text{-};\text{-},}\chi_0\mathtt{,}\iota_0\mathtt{,v_0)}$ will not satisfy all equations required of a CwD. It will satisfy 1–7 of Definition E:3 on page 142, but not for instance 8c), which would require us to identify some maps of $\mathtt{E_0}$. If one could identify a well behaved subcategory of $\mathtt{E_0}$, such as the environments in which each element is unique up to definitional equality, this quotient may be easier to carry out.

6 Future work

We have left quite a few questions open in this work. Aside from the obvious task of proving or disproving the conjectures, there are several ways one could wish to continue the work started here. We will name

a few, which come to mind, but which we have not worked out in any detail.

6.1 Rules which introduce equalities

The term systems presented here have rules which introduce terms. As we saw with E-categories we could emulate equations by introducing an equality sort, but it would be more satisfying to have "native" support for rules introducing equalities. One way this might be achieved is to introduce more elements in $\mathcal{E}$ `s` for each `s : [ℂ, Set]`. These new elements would represent pairs of compatible elements of some fibre of `s`, which will then be identified when applying `ext`.

Complications with this approach is that one might no longer have a unique rule introducing each subterm of a rule. One then finds one self in need of coherence rules ensuring that each de-facto equality of subterms is justified by applications of equality rules.

6.2 A weaker notion of homomorphism

As we saw in Remark E:57 the notion of homomorphism of `(`$\mathcal{R}$`,D)`-structures is quite strict and we could imagine weaker notions.

For example, one could, co-inductively or inductively, define two elements `a,a' : A x` as *"indistinguishable"* iff for each `b : A y` and non-identity `f : y → x`, such that `a = A f b`, there is a `b'`, indistinguishable from `b`, such that `a' = A f b'` and for every non-identity `g : y → z` either `g = f` or `A g b = A g b'`. A weaker notion of homomorphism would then only preserve operations up to indistinguishability.

6.3 Working out 2-categorical properties of CwDs

Categories with definitions form a 2-category, and an obvious line of investigation is to work out how different constructions of 2-categories apply to this specific case. For instance, one could suspect that whiskering gives a way to pull back homomorphisms between models along translations between theories. One could also work out what a monad on a CwD is.

References

Belo, João Filipe (2008). "Dependently Sorted Logic". In: *Types for Proofs and Programs: International Conference, TYPES 2007, Cividale des Friuli, Italy, May 2-5, 2007 Revised Selected Papers*. Ed. by Marino Miculan, Ivan Scagnetto, and Furio Honsell. Berlin, Heidelberg: Springer Berlin Heidelberg, pp. 33–50. ISBN: 978-3-540-68103-8. DOI: 10.1007/978-3-540-68103-8_3. URL: http://dx.doi.org/10.1007/978-3-540-68103-8_3.

Cartmell, John (1986). "Generalised algebraic theories and contextual categories". In: *Annals of Pure and Applied Logic* 32, pp. 209–243. ISSN: 0168-0072. DOI: http://dx.doi.org/10.1016/0168-0072(86)90053-9. URL: http://www.sciencedirect.com/science/article/pii/0168007286900539.

Harper, Robert and Frank Pfenning (2005). "On Equivalence and Canonical Forms in the LF Type Theory". In: *ACM Trans. Comput. Logic* 6.1, pp. 61–101. ISSN: 1529-3785. DOI: 10.1145/1042038.1042041. URL: http://doi.acm.org/10.1145/1042038.1042041.

Hofmann, Martin (1997). "Syntax and semantics of dependent types". In: *Semantics and logics of computation (Cambridge, 1995)*. Vol. 14. Publ. Newton Inst. Cambridge Univ. Press, Cambridge, pp. 79–130. DOI: 10.1017/CBO9780511526619.004. URL: http://dx.doi.org/10.1017/CBO9780511526619.004.

Makkai, Michael (1995). *First Order Logic with Dependent Sorts, with Applications to Category Theory, preliminary version Nov 6*. URL: http://www.math.mcgill.ca/makkai/folds/foldsinpdf/FOLDS.pdf.

Palmgren, Erik (2016). *Categories with families, FOLDS and logic enriched type theory*. arXiv: 1605.01586.

STOCKHOLM DISSERTATIONS IN MATHEMATICS

Dissertations at the Department of Mathematics Stockholm University

1–41. 2005–2012.

42. Lundqvist, Johannes: *On amoebas and multidimensional residues.* 2013.
43. Jost, Christine: *Topics in computational algebraic geometry and deformation quantization.* 2013.
44. Alexandersson, Per: *Combinatorial methods in complex analysis.* 2013.
45. Höhna, Sebastian: *Bayesian phylogenetic inference: Estimating diversification rates from reconstructed phylogenies.* 2013.
46. Aermark, Lior: *Hardy and spectral inequalities for a class of partial differential operators.* 2014.
47. Alm, Johan: *Universal algebraic structures on polyvector fields.* 2014.
48. Jafari-Mamaghani, Mehrdad: *A treatise on measuring Wiener–Granger causality.* 2014.
49. Martino, Ivan: *Ekedahl invariants, Veronese modules and linear recurrence varieties.* 2014.
50. Ekheden, Erland: *Catastrophe, ruin and death: Some perspectives on insurance mathematics.* 2014.
51. Johansson, Petter: *On the topology of the coamoeba.* 2014.
52. Lopes, Fabio: *Spatial marriage problems and epidemics.* 2014.
53. Bergh, Daniel: *Destackification and motivic classes of stacks.* 2014.
54. Olsson, Fredrik: *Inbreeding, effective population sizes and genetic differentiation: A mathematical analysis of structured populations.* 2015.
55. Forsgård, Jens: *Tropical aspects of real polynomials and hypergeometric functions.* 2015.
56. Tveiten, Ketil: *Period integrals and other direct images of D-modules.* 2015.
57. Leander, Madeleine: *Combinatorics of stable polynomials and correlation inequalities.* 2016.
58. Petersson, Mikael: *Perturbed discrete time stochastic models.* 2016.
59. Oneto, Alessandro: *Waring-type problems for polynomials: Algebra meets Geometry.* 2016.
60. Malmros, Jens: *Studies in respondent-driven sampling: Directed networks, epidemics, and random walks.* 2016.
61. Espíndola, Christian: *Achieving completeness: from constructive set theory to large cardinals.* 2016.
62. Bergvall, Olof: *Cohomology of arrangements and moduli spaces of curves.* 2016.
63. Backman, Theo: *Configuration spaces, props and wheel-free deformation quantization.* 2016.
64. Gylterud, Håkon Robbestad: *Univalent Types, Sets and Multisets.* 2017.

Distributor
Department of Mathematics
Stockholm University
SE-106 91 Stockholm, Sweden