# DEVELOPMENT OF GROUP THEORY IN THE LANGUAGE OF INTERNAL SET THEORY

A THESIS SUBMITTED TO THE UNIVERSITY OF MANCHESTER
FOR THE DEGREE OF DOCTOR OF PHILOSOPHY
IN THE FACULTY OF SCIENCE AND ENGINEERING

2019

Zoltan A. Kocsis

School of Natural Sciences
Department of Mathematics

# Contents

**Word Count: 27140**

# List of Tables

# List of Figures

# Abstract

This thesis explores two novel algebraic applications of Internal Set Theory (IST). We propose an explicitly topological formalism of structural approximation of groups, generalizing previous work by Gordon and Zilber. Using the new formalism, we prove that every profinite group admits a finite approximation in the sense of Zilber. Our main result states that well-behaved actions of the approximating group on a compact manifold give rise to similarly well-behaved actions of periodic subgroups of the approximated group on the same manifold. The theorem generalizes earlier results on discrete circle actions, and gives partial non-approximability results for SO(3). Motivated by the extraction of computational bounds from proofs in a “pure” fragment of IST (Sanders), we devise a “pure” presentation of sheaves over topological spaces in the style of Robinson and prove it equivalent to the usual definition over standard objects. We introduce a non-standard extension of Martin-Löf Type Theory with a hierarchy of universes for external propositions along with an external standardness predicate, allowing us to computer-verify our main result using the Agda proof assistant.

# Declaration

No portion of the work referred to in this thesis has been submitted in support of an application for another degree or qualification of this or any other university or other institute of learning.

# Copyright

# Acknowledgements

First and foremost, I wish to thank my supervisor, Prof. Alexandre Borovik, for taking me on as a student, for guiding me through the academic process, for providing mathematical advice, and for clearing up all the mundane and bureacratic obstacles that got in my way. Special thanks to

- My academic sibling Ulla Karhumäki and my officemate Jacob Cable, for the countless hours of productive mathematical discussion.
- My teachers and mentors: Péter Diviánszky, József Farkas, Marwan Fayed, Viola Somogyi, Jerry Swan, and all others who taught me mathematics. This work would not exist without them.
- My colleagues Nataliya Balabanova, Jacob Cable, Mahah Javed, Elliot McKernon, Rob Nicolaides, Joseph Razavi and Jerry Swan for proof-reading and sanity-checking this document. Naturally, I am responsible for any remaining errors.
- My family and friends, for their love and support.

# Chapter 1

# Introduction

Following Robinson's introduction of nonstandard analysis (via ultrafilter constructions of nonstandard models), Nelson [32] developed an axiomatic set theory (Internal Set Theory, IST) that extends the familiar Zermelo-Fraenkel Set Theory, and serves as a convenient framework for the practice of nonstandard analysis. Here we present two novel applications of Internal Set Theory to algebra.

Chapter 1 gives a concise, self-contained introduction to the theory and practice of Internal Set Theory, with a particular focus on doing topology in the non-standard setting via the formalism of (what we call) predicated spaces. Most results presented in this chapter are well-known and have appeared in the literature in various forms; the novelty resides in our presentation, which emphasizes the analogies between the theory of Alexandroff spaces in ordinary set theory and the theory of general topological spaces in Internal Set Theory.

The results in Chapter 2 concern structural approximations of groups, in the sense of Zilber [50, 51]. Using Internal Set Theory, we propose a new notion of approximation that incorporates an explicit topological ingredient and includes both Zilber's notion of finite approximation and Gordon's [1] notion of LEF group as special cases. Using the new language, we prove that any profinite group admits a finite approximation in the sense of Zilber (Proposition 2.2.18). We introduce the notion of Alexandroff approximation, and show that the class of groups admitting Alexandroff finite approximations coincides with the class of locally finite groups (Proposition 2.2.36). Our main result (Theorem 2.3.9) is as follows: if a group $H$ approximates $G$, then well-behaved actions of $H$ on a compact manifold $M$ give rise to similarly well-behaved actions of periodic subgroups of $G$ on the same manifold $M$. As a corollary of Theorem 2.3.9, we obtain partial results about the non-approximability of $SO(3)$ in the new formalism (Theorem 2.4.10).

Chapter 3 contains two shorter results. Inspired by the recent ultrapower proof of the monotone subsequence theorem due to Baszczyk, Kanovei, Katz and Nowik [5], we give a straightforward, ultrapower-free proof using Internal Set Theory. Motivated by the work of S. Sanders on extracting computational bounds from proofs in a "pure" fragment of Internal Set Theory, we give a novel, "pure" presentation of sheaves (Definition 3.2.9) over topological spaces in the style of Robinson's characterization of continuity, and prove it equivalent to the usual definition for standard objects (Theorem 3.2.15, Proposition 3.2.17).

In Chapter 4 we present a non-standard variant of Martin-Löf Type Theory that relates to ordinary Martin-Löf Type Theory the same way Internal Set Theory does to usual Zermelo-Fraenkel Set Theory. Our extended type theory has a hierarchy of universes for external propositions along with an external standardness predicate, allowing us to translate our proof of Theorem 2.3.9 into a type-theory setting, and computer-verify the resulting proof script using the Agda proof assistant.

## 1.1 The road to Internal Set Theory

**1.1.1.** In this section we introduce the axioms of Internal Set Theory, and explain its relationship to usual (ZFC) set theory. We presume familiarity with the terminology of first-order logic (languages, theories, free and bound variables, Hilbert-style proof systems, prenex forms) and the elementary axiomatics of Zermelo-Fraenkel Set Theory, to the extent covered in the first two chapters of Lévy's *Basic Set Theory* [29]. For a gentler introduction to Internal Set Theory, we recommend Robert's *Nonstandard Analysis* [40].

## Logic

**1.1.2.** We fix the notation for the logical connectives as $\neg$ (negation), $\vee$ (disjunction), $\wedge$ (conjunction) and $\rightarrow$ (implication). Notations such as $\sim, \supset, \&$ never stand for connectives, and may appear in the text with other (non-logical) meaning. We strive to make economical use of parentheses. In particular, we often write implication chains $\varphi_1 \rightarrow (\varphi_2 \rightarrow \varphi_3)$ as $\varphi_1 \rightarrow \varphi_2 \rightarrow \varphi_3$.

**1.1.3.** There are three major proof calculi for the first-order predicate calculus. Structural proof theory is best done in terms of Gentzen-style Sequent Calculus, which uncovers all the deep symmetries of logic. Prawitz's calculus of Natural Deduction displays no particularly good proof-theoretic behavior, at least for classical logic, but it corresponds closely to how we write proofs in *mathematics*. Finally, Hilbert-style proof calculi are appropriate for certain proof translation arguments. Nelson [33] uses a Hilbert-style system for his meta-theoretic results on Internal Set Theory.

**1.1.4.** Hilbert-style systems have only one inference rule: *modus ponens*, from $\varphi$ and $\varphi \rightarrow \psi$ infer $\psi$. When translating from one language to another, one needs to verify that the translations of the axioms are provable and that the rules of inference are preserved under translation, so having only one rule of inference proves to be a huge convenience. The logical axioms of our Hilbert system have the following forms:

- **K**: $\varphi_1 \rightarrow \varphi_2 \rightarrow \varphi_1$,
- **S**: $(\varphi_1 \rightarrow \varphi_2 \rightarrow \varphi_3) \rightarrow (\varphi_1 \rightarrow \varphi_2) \rightarrow \varphi_1 \rightarrow \varphi_3$,
- **N**: $(\neg\varphi_1 \rightarrow \neg\varphi_2) \rightarrow (\neg\varphi_1 \rightarrow \varphi_2) \rightarrow \varphi_1$,
- **U1**: $(\forall x.\phi(x)) \rightarrow \phi(t)$ where $t$ is any constant or variable symbol,
- **U2**: $(\forall x.\varphi_1 \rightarrow \varphi_2) \rightarrow \varphi_1 \rightarrow \forall y.\varphi_2$ where $x$ does not occur in $\varphi_1$,
- **U3**: $(\forall x.\varphi_1 \rightarrow \varphi_2) \rightarrow (\forall x.\varphi_1) \rightarrow \forall x.\varphi_2$

**1.1.5.** The first two axioms, **K** and **S**, play structural roles, enunciating the logical principles *weakening* and *contraction*. The third axiom, **N**, represents *reductio ad absurdum*, controlling the behavior of $\neg$ and expressing that the logic under consideration is classical (as opposed to e.g. intuitionistic), while the last three govern the meaning of the universal quantifier.

**1.1.6.** Apart from the usual symbols and predicates, our first-order languages often contain the special-purpose unary predicate *st*. Normally, we read $\mathrm{st}(x)$ as *$x$ is standard*. The "external" quantifiers $\forall^{st}, \exists^{st}$ are defined as abbreviations, with $\forall^{st} x.\phi(x)$ and $\exists^{st} x.\phi(x)$ abbreviating $\forall x.\mathrm{st}(x) \to \phi(x)$ and $\exists x.\mathrm{st}(x) \wedge \phi(x)$ respectively.

**1.1.7.** The superscript $Q^{\mathrm{fin}}$ indicates the finiteness of the object introduced by the quantifier $Q$. In particular, we can take it to abbreviate any of the usual definitions of *finite set* in ZFC Set Theory. However, we need to exercise caution when considering set theories that do not take the Axiom of Choice, as results such as Theorem 1.2.5 would depend on which of the many (not provably equivalent without Choice) definitions of finiteness we choose.

**1.1.8.** Throughout this document, we maintain a strict distinction between *relations* and proper *predicates*: we reserve the use of the former term to indicate predicates that are represented by sets, as subset of a Cartesian product (e.g. the order relation $<$ on the natural numbers is represented by the set $\left\{(x, y) \in \mathbb{N}^2 \,\middle|\, x < y\right\}$), while the term *$n$-ary predicate* refers to formulae with $n$ free variables in the language under consideration. The reader is already familiar with a proper binary predicate that does not constitute a relation in this sense: the global membership predicate $\in$ of Zermelo-Fraenkel Set Theory.

## Adjoining Ideal Elements

**1.1.9. Definition.** The *language of the theory* **PAK** consists of the language of Peano Arithmetic extended with a formal constant symbol $K$. The *theory* **PAK** consists of the axioms of Peano Arithmetic, and the following axioms (one for each natural number):

K0 $0 < K$,

K1 $1 < K$,

K2 $2 < K$,

$\dots$

Kn $n < K$,

$\dots$

**1.1.10.** Notice that the theory **PAK** does not extend Peano Arithmetic with new induction axioms. The induction axiom $\varphi(0) \wedge (\forall n.\varphi(n) \to \varphi(n+1)) \to \forall x.\varphi(x)$ belongs to **PAK** only if we choose $\varphi$ from among the formulae in the language of Peano Arithmetic. In particular, $0 < K \wedge (\forall n.n < K \to n+1 < K) \to \forall x.x < K$ does not belong to the *axioms* of **PAK**, since $\varphi(n) \leftrightarrow n < K$ contains the constant symbol $K$, which lives outside the language of Peano Arithmetic. However, we can prove that we *do* have $\forall y.0 < y \wedge (\forall n.n < y \to n+1 < y) \to \forall x.x < y$ among the *theorems* of Peano Arithmetic (use inducton on the formula $\forall y.0 < y \wedge (\forall n.n < y \to n+1 < y) \to x < y$, exercise!), and hence among the theorems of **PAK**. Substituting $y = K$ using the axiom **U1** realizes the previous non-axiom as a theorem of **PAK**!

**1.1.11.** Whenever Peano Arithmetic proves that all numbers $n$ have a given property $\varphi$, **PAK** proves that $K$ has the property $\varphi$. This follows immediately from the fact that the axioms of Peano Arithmetic form a proper subset of the axioms of the theory **PAK**. We will shortly prove a converse of this observation: whenever **PAK** proves that $K$ has some property $\varphi(K)$, and one can write this property $\varphi(-)$ in the language of Peano Arithmetic, then we can find a number $n \in \mathbb{N}$ such that $\varphi(n)$ holds (and in that case Peano Arithmetic proves $\varphi(n)$).

**1.1.12. Definition.** Consider theories $T_1, T_2$ such that the language of $T_1$ forms a proper subset of the language of $T_2$. We call $T_2$ a *conservative extension* of $T_1$ if for every sentence $\varphi$ in the language of $T_1$, we have $T_1 \vdash \varphi$ whenever $T_2 \vdash \varphi$.

**1.1.13. Proposition.** The theory **PAK** is a conservative extension of Peano Arithmetic.

**Proof.** Consider a sentence $\varphi$ in the language of Peano Arithmetic, and assume that **PAK** proves $\varphi$. Take any **PAK**-proof of $\varphi$: such a proof invokes finitely many of the axioms of **PAK**, in particular we can find a largest number $n \in \mathbb{N}$ such that the proof uses the axiom Kn. Replace all occurrences of $K$ in the proof with $n+1$, and all the axioms Ki with Peano Arithmetic proofs of $i < n+1$. This cannot fail: by the maximality of $n$, we have $i < n < n+1$ for all $i$ that occur in the proof. Doing all the replacements yields a proof of the sentence $\varphi$ in Peano Arithmetic.
**Qed.**

**1.1.14. Corollary.** Consider a formula $\varphi(-)$ of Peano Arithmetic. If **PAK** proves $\varphi(K)$, then Peano Arithmetic proves $\varphi(n)$ for some numeral $n \in \mathbb{N}$.

**Proof.** Apply the algorithm of Proposition 1.1.13.

**Qed.**

**1.1.15.** Notice that Corollary 1.1.14 relies on the algorithm described in the proof of Proposition 1.1.13, and not merely on the statement of the proposition: an instance of *proof relevance* in mathematics. We use the term *corollary* in this proof-relevant sense throughout our work.

**1.1.16.** The construction of **PAK** privileges the relation $<$ over other possible relations. Indeed, we could have defined a theory **PAKd** in the language of **PAK** that has the axioms of Peano Arithmetic, along with the following axioms (one for each natural):

K1 1 divides $K$,

K2 2 divides $K$,

...

Kn $n$ divides $K$,

...

and the resulting theory would satisfy the analogue of Proposition 1.1.13, if one replaced $K$ with the product $\prod_{i<n} i$ instead of $n+1$.

**1.1.17.** The constant symbol $K$ of **PAK** behaves like an ideal element with respect to the order relation, and that of **PAKd** behaves ideally with respect to divisibility. One should be able to extend the method of paragraph 1.1.16 to add new constants that behave like ideal elements for any relation, as long as such ideal elements can coexist with Peano Arithmetic in the sense that finitely many of the new axioms do not contradict Peano Arithmetic. The notion of admissibility (Definition 1.1.18) formalizes this intuition.

**1.1.18. Definition.** We call a binary predicate $R(-,-)$ in the language of Peano Arithmetic *admissible* if for any finite subset of the natural numbers $F$ we can find $y$ such that Peano Arithmetic proves that $R(x,y)$ holds for all $x \in F$.

**1.1.19. Proposition.** Every admissible binary predicate $R(-,-)$ gives rise to a theory **PAKR** in the language of **PAK** conservatively extending Peano Arithmetic with ideal elements for $R$. Vice versa, if a binary predicate gives rise to such a conservative extension, we can conclude the admissibility of $R(-,-)$.

**Proof.** We leave the forward direction as an exercise to the reader. For the backward direction, consider any finite set $F \subseteq \mathbb{N}$. The theory **PAKR** proves the conjunction $\bigwedge_{x\in F} R(x,K)$ (since it proves each of the axioms Ki). Hence, **PAKR** also proves $\exists y. \bigwedge_{x\in F} R(x,y)$, a sentence of Peano Arithmetic. By conservative extension, Peano Arithmetic proves $\exists y. \bigwedge_{x\in F} R(x,y)$, so it proves that $R(x,y)$ holds for all $x \in F$. Using Definition 1.1.18, we conclude the admissibility of $R$.
**Qed.**

**1.1.20.** As we have seen, **PAK**-style extensions add new constants for ideal numbers, but do not otherwise change Peano Arithmetic. However, we cannot quantify over (or otherwise keep track of) these additions at the level of syntax. Hence, we could increase the expressiveness of our extensions by including an explicit new predicate for those elements that Peano Arithmetic already had, even before we performed the idealization.

**1.1.21. Definition.** Consider an admissible predicate $R(-,-)$. The *language of the theory* **PAKS** consists of the language of Peano Arithmetic extended with a formal constant symbol $K$ and a unary atomic predicate $\mathrm{st}(-)$. The *theory* **PAKSR** consists of the axioms of Peano Arithmetic, along with the three new axioms below:

1. $\forall x.\mathrm{st}(x) \to R(x,K)$,
2. $\mathrm{st}(0)$,
3. $\forall n.\mathrm{st}(n) \to \mathrm{st}(n+1)$.

**1.1.22.** Similarly to 1.1.10, our construction of **PAKSR** does not add new induction axioms to Peano Arithmetic. In particular, **PAKSR** does not prove $\forall x.\mathrm{st}(x)$, even though it proves both $\mathrm{st}(0)$ and $\forall n.\mathrm{st}(n) \to \mathrm{st}(n+1)$. Notice that $\neg\mathrm{st}(K)$ does not occur among the axioms.

**1.1.23. Exercise.** Prove $\neg\mathrm{st}(K)$ in **PAKSR** for $R(x,y) \leftrightarrow x < y$. Choose carefully another binary predicate $R$ in such a way that you can prove $\mathrm{st}(K)$ in the corresponding theory **PAKSR**.

**1.1.24.** Instead of constructing a new theory **PAKSR** for each relation $R$, we could parallelize our construction, extending Peano Arithmetic simultaneously with all possible ideal elements. Indeed, as the construction of the theory **PAKSR** extends Peano Arithmetic with ideal elements, so will the *Idealization* axiom of Internal Set Theory create ideal elements with respect to any admissible relation. As such, we will not spend time proving conservative extension over Peano Arithmetic for the likes of **PAKSR**: the proof of the conservative extension theorem for Internal Set Theory over ZFC Set Theory supersedes such results anyway, and we sketch the main ingredient of that latter proof below.

## Internal Set Theory

**1.1.25. Definition.** The *language of Internal Set Theory* consists of a binary predicate symbol $- \in -$ (membership) and a unary predicate symbol $\mathrm{st}(-)$. The first-order theory referred to as *Internal Set Theory* consists of the axioms of Zermelo-Fraenkel Set Theory, the Axiom of Choice, and the additional axiom schemata Idealization, Standardization and Transfer defined below.

**1.1.26.** Given a set $A$, we introduce the following abbreviated quantifiers:

- $\forall^{st} x \in A. \dots$ abbreviates $\forall x. x \in A \wedge \mathrm{st}(x) \to \dots,$
- $\exists^{st} x \in A. \dots$ abbreviates $\exists x. x \in A \wedge \mathrm{st}(x) \wedge \dots,$

**1.1.27. Definition.** Consider a formula $\varphi$ in the language of Internal Set Theory. We call $\varphi$ an *internal formula* if it does not contain any occurrences of the predicate $\mathrm{st}(-)$. In accordance with the observations of sections 1.1.10 and 1.1.22, we shall permit internal formulae to contain parameters ranging over both standard and non-standard sets.

**1.1.28. Axiom Schema of Idealization:** Consider an internal formula $\varphi$, and a variable $\mathcal{F}$ fresh with respect to $\varphi$. The following statements are equivalent.

1. $\forall^{st\,\mathrm{fin}} \mathcal{F}. \exists y. \forall x \in \mathcal{F}. \varphi(x, y)$,
2. $\exists y. \forall^{st} x. \varphi(x, y)$

Notice that the first clause captures the notion of admissible relation introduced in Definition 1.1.18, and the schema internalizes the process of adjoining new constants for ideal numbers. For example, instantiating Idealization with the predicate $\varphi(x, y)$ abbreviating "$x, y \in \mathbb{N}$ and $x$ divides $y$" gives us an analogue of the ideal element $K \in \mathbb{N}$ that appears in 1.1.16.

**1.1.29. Axiom Schema of Transfer:** Consider an internal formula $\varphi$ with free variables $x_1, \dots, x_n$ and no others. The following holds:

$$\forall^{st} x_1. \dots \forall^{st} x_{n-1}. \left(\forall^{st} x_n. \varphi \to \forall x_n. \varphi\right)$$

That is, if a transfer property holds for every standard element of a standard set, then it holds for every element of that set[1].

**1.1.30. Axiom Schema of Standardization:** Take an arbitrary (internal or external) formula $\varphi$ with one free variable, and a standard set $G$. Then we can construct a standard set, denoted $\{\!\!\{ x \in G \mid \varphi(x) \}\!\!\}$ such that the following are equivalent for each element $a \in G$:

1. $a \in \{\!\!\{ x \in G \mid \varphi(x) \}\!\!\}$

2. $\mathrm{st}(a) \to \varphi(a)$

The notation closely resembles Comprehension: indeed, we can see this axiom schema as an external comprehension principle for a restricted class of formulae. Given the other axioms, one can show that the set constructed by Standardization is the unique set satisfying the property above.

**1.1.31.** Recall that the axiom schema of Comprehension,

$$\forall z.\exists! y.\forall x.x \in y \leftrightarrow (x \in z \land \varphi)$$

occurs as one of the axiom schemata of Zermelo-Fraenkel Set Theory. The axiom justifies the use of set builder notation, by writing $\{x \in z \mid \varphi\}$ for the set $y$ whose unique existence the schema asserts. However, Internal Set Theory does not add new instances of the Comprehension schema to the underlying ZFC Set Theory: as such, instances of set-builder notation where the formula $\varphi$ is not internal may fail to denote any set at all! As an example, take $\{x \in \mathbb{N} \mid \mathrm{st}(n)\}$. Internal Set Theory does not prove the existence of a set that contains precisely the standard naturals (indeed, we will see in Corollary 1.2.12 that it proves the *non-existence* of such a set). In a departure from usual mathematics, the practitioner of Internal Set Theory has to take great care to avoid such *illegal set formation*, by making sure that all instances of set builder notation use only internal formulae.

## Galactic Halo theorem

**1.1.32.** Nelson [33] gave a proof-theoretic algorithm which translates proofs in Internal Set Theory to proofs in ZFC. Theorem 1.1.39, the key ingredient of Nelson's argument,

[1] Cf. the Tarski-Vaught criterion for elementary substructures.

will make an appearance in the subsequent chapters. As such, we present a proof of Theorem 1.1.39. To prove conservative extension, one would have to further prove the admissibility of the modus ponens rule in translation, the preservation of the *logical axioms* (i.e. that one can indeed prove the translations of all instances of the logical schemata introduced in 1.1.4 purely inside ZFC), and (since the translation works only for bounded formulae) introduce and eliminate a "universe bound". Proving the preservation of the logical axioms requires a non-trivial use of the Tychonoff theorem (see [33]-Theorem 3). We assume a good working knowledge of Internal Set Theory in this subsection: readers who have not worked in Internal Set Theory before should feel free to skip this subsection for now and proceed directly to Section 1.2.

**1.1.33. Lemma.** Consider a standard set $V$ and an internal formula $\varphi$ of Internal Set Theory with two free variables $x, y$. Assume that $\forall^{st} x \in V . \exists^{st} y \in V . \varphi(x, y)$. Then IST proves the existence of a standard function $f : V \to V$ such that $\forall^{st} x \in V . \varphi(x, f(x))$.

**Proof.** Define the function $\overline{f} = \{\!\{ (x, Y) \in V \times \mathcal{P}(V) \mid Y = \{\!\{ y \in V \mid \varphi(x, y) \}\!\} \}\!\}$ via a nested use of the Standardization axiom. Since the set-valued function $\overline{f}$ never takes the value $\emptyset$, the Axiom of Choice gives a function $f : V \to V$ with the required property. **Qed.**

**1.1.34.** Notice that the proof of Lemma 1.1.33 does not rely the internality assumption for $\varphi$ in any way. However, the lemma, even when restricted to internal formulae, allows us to prove all instances (internal or external) of Standardization.

**1.1.35. Exercise.** Show that the theory obtained by adding the Idealization, Transfer principles and Lemma 1.1.33 to Zermelo-Fraenkel Set Theory with the Axiom of Choice proves every instance of the Standardization schema. Hint: Use Theorem 1.1.39 twice.

**1.1.36. Definition.** Given a set $S$, we denote its *finite powerset*, the set of all its finite subsets, by $\mathcal{P}^{\mathrm{fin}}(S)$. However, if the current context has a formal constant $U$ standing in as a bound for the universe of discourse, we treat the formula $S \in \mathcal{P}^{\mathrm{fin}}(U)$ as an abbreviation for the sentence "$S$ forms a finite set".

**1.1.37.** For the sake of simplicity, we may leave bounds implicit, but assume that every quantifier has a standard bound in the next few paragraphs. Note that Kanovei [26] shows that the unbounded sentence $\forall F . (\forall^{st} n \in \mathbb{N} . \mathrm{st}(F(n)) \to \exists^{st} G . \forall^{st} n \in \mathbb{N} . F(n) = G(n))$ is not equivalent to any sentence of ZFC (and is therefore independent of IST).

**1.1.38. Definition.** Consider bounded sentences $\Psi_1, \Psi_2$ of Internal Set Theory, where $\Psi_2$ has the form $\forall^{st} x_1.\forall^{st} x_2.\ldots\exists^{st} y_1.\exists^{st} y_2.\ldots\psi$ for some internal formula $\psi$. We write $[\Psi_1] = \Psi_2$ and say that $\Psi_1$ has *Nelson normal form* $\Psi_2$ if we can construct a proof tree with conclusion labeled by $[\Psi_1] = \Psi_2$ using finitely many instances of the following rules (for more about proof trees see 4.1.15).

$$\frac{}{[\mathrm{st}(x)] = (\exists^{st} q.x = q)}\ \mathrm{st}$$

or

$$\frac{}{[\varphi] = \varphi}\ \mathrm{int}$$

or

$$\frac{[\Phi] = \forall^{st} x_1 \in A_1.\ldots\exists^{st} y_1 \in E_1.\ldots\varphi}{[\neg\Phi] = \forall^{st}\overline{y}_1 \in \prod_i A_i \to E_1.\ldots\exists^{st} x_1 \in A_1.\ldots\neg\varphi[y_i := \overline{y}_i(x_1,\ldots)]}\ \neg$$

or

$$\frac{[\Phi] = \forall^{st} x_1 \in A_1.\ldots\exists^{st} y_1 \in E_1.\ldots\varphi}{[\forall^{st} z \in A.\Phi] = \forall^{st} z \in A.\forall^{st} x_1 \in A_1.\ldots\exists^{st} y_1 \in E_1.\ldots\varphi}\ \forall^{st}$$

or

$$\frac{[\Phi] = \forall^{st} x_1 \in A_1.\ldots\exists^{st} y_1 \in E_1.\ldots\varphi}{[\forall z.\Phi] = \forall^{st} x_1 \in A_1.\ldots\exists^{st} Y_1 \in \mathcal{P}^{\mathrm{fin}}(E_1).\ldots\forall z.\exists y_1 \in Y_1.\ldots\varphi}\ \forall$$

or

$$\frac{[\Phi_1] = \forall^{st} x_1 \in A_1.\ldots\exists^{st} y_1 \in E_1.\ldots\varphi_1 \qquad [\Phi_2] = \forall^{st} v_1 \in B_1.\ldots\exists^{st} w_1 \in F_1.\ldots\varphi_2}{[\Phi_1 \to \Phi_2] = \forall^{st} v_1 \in B_1.\ldots\forall^{st}\overline{y}_1 \in \prod_i A_i \to E_1.\ldots\exists^{st} x_1 \in A_1.\ldots\exists^{st} w_1 \in F_1.\ldots\varphi'}\ \to$$

where $\varphi'$ abbreviates $\varphi_1[y_i := \overline{y}_i(x_1,\ldots)] \to \varphi_2$, $\Phi, \Phi_2$ stand for arbitrary non-internal formulae, $\Phi_1$ stands for an arbitrary formula, $\varphi_i$ stand for internal formulae, variable $z$ occurs free in each $\Phi_i$ and we choose $q$ as a fresh variable.

**1.1.39. Theorem** (Galactic Halo[2])**.** Every bounded sentence $\Psi$ of Internal Set Theory has a logically equivalent (modulo theory), unique Nelson normal form $[\Psi]$.

**Proof.** For uniqueness, observe that in Definition 1.1.38, the principal connective of $\Psi_1$ uniquely determines the next available rule of the proof tree. For existence, observe that the depth of the formula decreases with each application of a rule. Hence, one will eventually reach either an internal formula (which has itself as a unique Nelson normal form) or $\mathrm{st}(x)$ (which has Nelson normal form $\exists^{st} q.x = q$ for some fresh variable $q$). For logical equivalence, we argue by induction on structure of the formula, using the uniqueness and existence of the tree itself as a guide.

[2]Theorem 1 of [33]. Some authors call formulae of the form $\forall^{st} x.\varphi$ *halic*, and those of the form $\exists^{st} x.\varphi$ *galactic*. In naming this theorem, we pay homage to them.

1. Cases st,int,$\forall^{st}$: Follow immediately from the definitions.

2. Case $\neg$: We have

$$\begin{aligned}
\neg\Phi &\leftrightarrow \neg[\Phi] && \text{by inductive assumption}\\
&\leftrightarrow \neg\forall^{st}x_1 \in A_1 \ldots \exists^{st}y_1 \in E_1 \ldots \varphi && \text{by the } \neg \text{ rule (premise)}\\
&\leftrightarrow \exists^{st}x_1 \in A_1 \ldots \forall^{st}y_1 \in E_1 \ldots \neg\varphi && \text{by de Morgan's laws}\\
&\leftrightarrow \forall^{st}\overline{y}_1 \in \Pi_i A_i \to E_1 \ldots \exists^{st}x_1 \in A_1 \ldots \varphi' && \text{by Lemma 1.1.33}\\
&\leftrightarrow [\neg\Phi]. && \text{by the } \neg \text{ rule}
\end{aligned}$$

where $\varphi'$ abbreviates $\neg\varphi[y_i := \overline{y}_i(x_1,\ldots)]$.

3. Case $\forall$: We have

$$\begin{aligned}
\forall z.\Phi &\leftrightarrow \forall z.[\Phi] && \text{by induction}\\
&\leftrightarrow \forall z.\forall^{st}x_1 \in A_1 \ldots \exists^{st}y_1 \in E_1 \ldots \varphi && \text{by the } \forall \text{ rule}\\
&\leftrightarrow \forall^{st}x_1 \in A_1 \ldots \forall z.\exists^{st}y_1 \in E_1 \ldots \varphi && \text{by quantifier switch}\\
&\leftrightarrow \forall^{st}x_1 \in A_1 \ldots \exists^{st}Y_1 \in \mathcal{P}^{\mathrm{fin}}(E_1) \ldots \varphi' && \text{by Idealization}\\
&\leftrightarrow [\forall z.\Phi]. && \text{by the } \forall \text{ rule}
\end{aligned}$$

where $\varphi'$ abbreviates $\forall z.\exists y_1 \in Y_1.\varphi$.

4. Case $\to$: We have

$$\begin{aligned}
\Phi' &\leftrightarrow ([\Phi_1] \to [\Phi_2]) && \text{by induction}\\
&\leftrightarrow (\neg[\Phi_1] \vee [\Phi_2]) && \text{by classical logic}\\
&\leftrightarrow ([\neg\Phi_1] \vee [\Phi_2]) && \text{by case } \neg \text{ above}\\
&\leftrightarrow ((\forall^{st}\overline{y}_1 \in \Pi_i A_i \to E_1 \ldots \exists^{st}x_1 \in A_1 \ldots \varphi') \vee\\
&\quad \forall^{st}v_1 \in B_1 \ldots \exists^{st}w_1 \in F_1 \ldots \varphi_2) && \text{by the } \neg \text{ rule}\\
&\leftrightarrow \forall^{st}v_1 \in B_1 \ldots \forall^{st}\overline{y}_1 \in \Pi_i A_i \to E_1 \ldots\\
&\quad \exists^{st}x_1 \in A_1 \ldots \exists^{st}w_1 \in F_1 \ldots \varphi' \to \varphi_2 && \text{by quantifier switch}\\
&\leftrightarrow [\Phi_1 \to \Phi_2]. && \text{by the } \to \text{ rule}
\end{aligned}$$

where $\Phi'$ abbreviates $\Phi_1 \to \Phi_2$ and $\varphi'$ abbreviates $\neg\varphi[y_i := \overline{y}_i(x_1,\ldots)]$.

This proves that every bounded formula of Internal Set Theory has a logically equivalent, unique Nelson normal form.

**Qed.**

**1.1.40.** Using the Galactic Halo theorem, Nelson gives a translation that converts proofs in Internal Set Theory to proofs in Zermelo-Fraenkel Set Theory with the Axiom of Choice. If a ZFC formula occurs as the conclusion of the proof inside Internal Set Theory, the resulting ZFC proof will retain the same conclusion, thus ensuring conservativity of Internal Set Theory over ZFC.

**1.1.41. Definition.** Given a sentence $\Phi$ of Internal Set Theory with Nelson normal form $[\Phi] = \forall^{st} x_1.\forall^{st} x_2.\dots \exists^{st} y_1.\exists^{st} y_2.\dots \varphi$, we refer to the ZFC sentence $\overline{[\Phi]} = \forall x_1.\forall x_2.\dots \exists y_1.\exists y_2.\dots \varphi$ as the *Nelson reduction of the formula* $\Phi$.

**1.1.42. Proposition.** We have an equivalence between every bounded sentence $\Phi$ of Internal Set Theory and its Nelson reduction $\overline{[\Phi]}$ when we interpret the latter as a sentence of Internal Set Theory.

**Proof.** Use Transfer.
**Qed.**

**1.1.43. Proposition** ([33]-Theorem 2)**.** Applying the Nelson reduction to Idealization, (simple) Standardization and Transfer axioms yields theorems of ZFC set theory.

**Proof.** For notational convenience, we omit most bounds and merge all consecutive quantifiers of the same sort into one quantifier, e.g. $\forall^{st} t$ abbreviates $\forall^{st} t_1.\forall^{st} t_2.\dots$ in what follows. Similarly for displayed variables in predicates and terms in substitutions. Our version of the proof differs from Nelson's account in the treatment of Idealization.

1. *Transfer*: A Transfer axiom takes the form $\forall^{st} t.(\forall^{st} x.\varphi(x,t)) \to \forall^{y}.\varphi(y,t)$ for an internal formula $\varphi$. We calculate its Nelson normal form as

$$\begin{aligned} & [\forall^{st} t.(\forall^{st} x.\varphi(x,t)) \to \forall^{y}.\varphi(y,t)] \\ = {} & \forall^{st} t.[(\forall^{st} x.\varphi(x,t)) \to \forall^{y}.\varphi(y,t)] \\ = {} & \forall^{st} t.\forall^{st} y.\exists^{st} x.\varphi(x,t) \to \varphi(y,t). \end{aligned}$$

   This gives rise to the Nelson reduction $\forall t.\forall y.\exists x.\varphi(x,t) \to \varphi(y,t)$, a tautology.

2. *Idealization F*: In the following, we use a purely formal placeholder constant $U$ for the universe. Recall that an Idealization axiom takes the form

$$(\forall^{st} B \in \mathcal{P}^{\mathrm{fin}}(U).\exists a.\forall b \in B.\varphi(a,b,w)) \to \exists x.\forall^{st} y.\varphi(x,y,w)$$

for any internal formula $\varphi$. We first compute the Nelson normal form of the right hand side: $[\exists x.\forall^{st} y.\varphi(x,y,w)] = \forall^{st} Y \in \mathcal{P}^{\mathrm{fin}}(U).\exists x.\forall y \in Y.\varphi(x,y,w)$. Using this, we get the Nelson normal form

$$\begin{aligned}&\forall Y \in \mathcal{P}^{\mathrm{fin}}(U).\exists B' \in \mathcal{P}^{\mathrm{fin}}(\mathcal{P}^{\mathrm{fin}}(U)).\forall w.\exists B \in B'.\\ &(\exists a.\forall b \in B.\varphi(a,b,w)) \to \exists x.\forall y \in Y.\varphi(x,y,w),\end{aligned}$$

which we need to prove in ZFC Set Theory. To do that, first notice its equivalence to

$$\begin{aligned}&\forall Y \in \mathcal{P}^{\mathrm{fin}}(U).\exists Z \in \mathcal{P}^{\mathrm{fin}}(U).\forall w.\\ &(\exists a.\forall z \in Z.\varphi(a,z,w)) \to \exists x.\forall y \in Y.\varphi(x,y,w),\end{aligned}$$

obtained by assuming the former then setting $Z = \bigcup B'$ to conclude the latter; then notice that we can write the latter as an instance of a logical tautology.

3. *Idealization B*: The backward Idealization axiom takes the form

$$(\exists x.\forall^{st} y.\varphi(x,y,w)) \to \forall^{st} B \in \mathcal{P}^{\mathrm{fin}}(U).\exists a.\forall b \in B.\varphi(a,b,w)$$

for any internal formula $\varphi$. Computing the Nelson normal form, we get

$$\begin{aligned}&\forall B \in \mathcal{P}^{\mathrm{fin}}(U).\exists Y' \in \mathcal{P}^{\mathrm{fin}}(\mathcal{P}^{\mathrm{fin}}(U)).\forall w.\exists Y \in Y'.\\ &(\exists x.\forall y \in Y.\varphi(x,y,w)) \to \exists a.\forall b \in B.\varphi(a,b,w),\end{aligned}$$

which coincides with the forward case up to renaming.

4. *Standardization*: We deal only with Standardization in the form of Lemma 1.1.33, which we can write as

$$\forall w.(\forall^{st} x.\exists^{st} y.\varphi(x,y,w)) \to \exists^{st} b : U \to U.\forall^{st} a.\varphi(a,b(a),w)$$

where $\varphi$ denotes an internal formula, as usual. We first calculate the Nelson normal form

$$\begin{aligned}&[\exists^{st} b : U \to U.\forall^{st} a.\varphi(a,b(a),w)] =\\ &\forall^{st} a : U \to U.\exists^{st} b.\varphi(a(b),b(a(b)),w).\end{aligned}$$

Now, we can take the Nelson reduction of the implication, and get

$$\forall a.\forall y.\exists x.\exists b.\varphi(x, y(x), w) \to \varphi(a(b), b(a(b)), w)$$

Finally, we add the universal quantification to obtain the monstrous formula

$$\begin{aligned}&\forall a.\forall y.\\ &\exists X.\exists B.\\ &\forall w.\exists x \in X.\exists b \in B.\varphi(x, y(x), w) \to \varphi(a(b), b(a(b)), w).\end{aligned}$$

Notice that we can prove the Nelson reduction of the formula above simply by setting $X = \{a(y)\}$, $B = \{y\}$. These fix $b$ and $x$ uniquely, and we get the goal

$$\begin{aligned}&\forall a.\forall y.\\ &\forall w.\varphi(a(y), y(a(y)), w) \to \varphi(a(y), y(a(y)), w),\end{aligned}$$

an instance of a logical tautology.

**Qed.**

**1.1.44.** The translation sketched above relies heavily on the Axiom of Choice: every switch of quantifiers and every translation of a modus ponens rule hides a use of Choice, and one also has to prove that the translations of the logical axioms (when instantiated with external formulae) also yield theorems of ZFC Set Theory. Proving this for logical axioms of the form **U3** requires invocations of Tychonoff's theorem for products of finite sets.

**1.1.45. Theorem.** Both of the following statements imply each other in Zermelo-Fraenkel Set Theory (without the Axiom of Choice):

- **Finite Tychonoff Theorem:** The product of an indexed family of finite topological spaces satisfies compactness with respect to the product topology.
- **Ultrafilter Lemma:** Every filter on any set occurs as a subfilter of some ultrafilter.

**Proof.** Follows from [29]-Theorem 2.21.
**Qed.**

**1.1.46.** The Ultrafilter Lemma is independent of Zermelo-Fraenkel Set Theory [4]. With that in mind, and in light of Theorem 1.1.45, we shall see the inevitability of the phenomenon discussed in 1.1.44: Internal Set Theory conservatively extends ZFC Set Theory, but Internal Set Theory without the Axiom of Choice does not conservatively extend Zermelo-Fraenkel Set Theory without the Axiom of Choice. We will prove this in a subsequent section by deducing the Ultrafilter Lemma (Lemma 1.3.40) in IST without using Choice. Apart from the Finite Tychonoff Theorem, the Nelson translation also uses full Choice to switch quantifiers[3], so one cannot use it to show that IST without Choice conservatively extends Zermelo-Fraenkel Set Theory with the Ultrafilter Lemma. However, one can use a model-theoretic argument of Hrbacek [23] to prove this conservative extension result directly.

## 1.2 Working in IST

**1.2.1.** From here on we work inside Internal Set Theory. Consequently, all the theorems that follow are stated and proved in Internal Set Theory, unless otherwise noted.

**1.2.2. Proposition.** Consider an internal predicate $\varphi$ with standard parameters. Assume the existence of a nonstandard $x$ such that $\varphi(x)$ holds. Then we can also find a standard $y$ satisfying $\varphi$.

**Proof.** If the predicate $\varphi$ is internal, so is $\neg\varphi$. The implication $(\forall^{st} y.\neg\varphi(y)) \to \forall x.\neg\varphi(x)$ holds by Transfer. Taking the contrapositive, we get the implication $(\exists x.\varphi(x)) \to \exists^{st} y.\varphi(y)$. Now assume that we have a nonstandard $x$ such that $\varphi(x)$ holds. Then *a fortiori* we have an $x$ such that $\varphi(x)$ holds (we just "forget to mention" the nonstandardness of $x$). Hence, the previous implication immediately gives a standard $y$ satisfying $\varphi(y)$.
**Qed.**

**1.2.3.** As per Proposition 1.2.2, we can use transfer on any internal formula: any internal formula has a prenex normal form $\forall x.Q_2 y.\dots\varphi(x,y)$ where $Q_i$ are quantifiers $\forall$ or $\exists$, and $\varphi$ is internal and quantifier-free. If $Q_1 = \forall$, then the Transfer axiom yields $\forall x.Q_2 y.\dots\varphi(x,y) \leftrightarrow \forall^{st} x.Q_2 y.\dots\varphi(x,y)$. Otherwise, Proposition 1.2.2 yields $\exists^{st} x.Q_2 y.\dots\varphi(x,y) \leftrightarrow \exists x.Q_2 y.\dots\varphi(x,y)$. Notice that we have already relied on this in

[3] However, this use of Choice turns out to be *innocent* in both Peano arithmetic with finite types and in the type theory presented in Chapter 4.

the proof of Theorem 1.1.39. More importantly, Proposition 1.2.2 allows us to make *provisional assumptions of standardness*: whenever we wish to prove an internal implication $\varphi \to \psi$, we can start the proof by assuming the standardness of all objects mentioned in $\varphi$. After we prove the conclusion $\psi$, we can freely discharge these standardness assumptions. In many situations, the additional standardness assumptions make the conclusion easier to prove, by enabling the use of previously constructed ideal elements. The reader will see a substantial example of the phenomenon "in action" in the proof of Theorem 1.2.7.

**1.2.4. Proposition.** Consider an internal formula $\psi$. The following statements are equivalent.

1. $\exists^{st\,\mathrm{fin}}\mathcal{F}.\forall y.\exists x \in \mathcal{F}.\psi(x, y)$,
2. $\forall y.\exists^{st} x.\psi(x, y)$.

**Proof.** Recall that for any internal $\varphi$, $\forall^{st\,\mathrm{fin}}\mathcal{F}.\exists y.\forall x \in \mathcal{F}.\varphi(x, y)$ and $\exists y.\forall^{st} x.\varphi(x, y)$ are equivalent. Since $\psi$ is an internal formula, so is $\neg\psi$. Setting $\varphi$ to $\neg\psi$, we get the logical equivalence of $\forall^{st\,\mathrm{fin}}\mathcal{F}.\exists y.\forall x \in \mathcal{F}.\neg\psi(x, y)$ and $\exists y.\forall^{st} x.\neg\psi(x, y)$. We can take the contrapositive and apply de Morgan's laws to obtain the equivalence of $\exists^{st\,\mathrm{fin}}\mathcal{F}.\forall y.\exists x \in \mathcal{F}.\psi(x, y)$ and $\forall y.\exists^{st} x.\psi(x, y)$ as desired.
**Qed.**

**1.2.5. Theorem.** Every element of a standard finite set is standard. Furthermore, if every element of a set $F$ is standard, then $F$ is finite.

**Proof.** First consider a standard finite set $F$. Then the following holds:

$$\exists^{st\,\mathrm{fin}} H.\forall x.\exists y \in H.x \in F \to x = y$$

simply by taking $H = F$. Applying Proposition 1.2.4, we obtain that $\forall y.\exists^{st} x.x \in F \to x = y$. But then $y$ is standard. Now consider a set $F$ whose elements are all standard. Then $\forall y.\exists^{st} x.x \in F \to x = y$. Applying the previous equivalence in reverse, we get that $F \subseteq H$ for some finite $H$. This proves that $F$ is finite.
**Qed.**

**1.2.6.** The proof of the combinatorialist's version of the compactness theorem, a well known theorem of de Bruijn and Erdős, provides an ideal entry point to Internal Set Theory, since it uses each of the new axioms at least once. The proof presented below hides only a compactness argument, but the general technique recurs very often, so we feel compelled to give an excruciatingly detailed proof. Ultimately, axiomatizing these situations will reveal a connection to Zilber's notion of structural approximation via Definition 2.2.3. In what follows, the word *graph* denotes an undirected graph with no loops or multi-edges. We denote the set of vertices of a graph $G$ as $V_G$, the set of edges as $E_G$. We call a graph finite if it contains finitely many vertices. A *coloring* of a graph consists of a map $f : V(G) \to C$ from the vertices of the graph to some set of colors $C$ such that no two vertices sharing the same edge get assigned the same color.

**1.2.7. Theorem** (de Bruijn-Erdős)**.** Consider a finite set of colors $C = \{1, \dots, k\}$. A graph $G$ admits a $C$-coloring precisely if every finite subgraph of $G$ admits a $C$-coloring.

**Proof.** One direction is obvious. For the other direction, assume that we can color every finite subgraph of $G$ with $k \in \mathbb{N}$ colors. In fact, since we can express our conclusion ($k$-colorability of $G$) using an internal formula, we can provisionally assume the standardness of both the graph $G$ and the finite set $C$ of colors. At the end, Transfer will eliminate these assumptions.

1. Given any finite set $N$ of vertices of $G$, we can find a finite subgraph of $G$ that contains all the vertices in $N$ (take the induced subgraph of the set $N$). The Idealization axiom applies to this situation: we get a finite subgraph $H \subseteq G$ that nevertheless contains every standard vertex of $G$.

2. Since $H$ is a finite subgraph of $G$, our assumption guarantees that it admits a $C$-coloring $f : H \to \{1, \dots k\}$. Identify the function $f$ with its graph, the set of all pairs $(v, c)$ such that $f(v) = c$. Then for any $v \in V_G$ we can find a unique $c \in \{1, \dots, k\}$ such that $(v, c) \in f$.

3. Use Standardization to define a standard set
$$f' = \{\!\![ (v,c) \in V_G \times \{1, \dots, k\} \,\big|\, (v,c) \in f ]\!\!\}.$$
We shall prove that the set $f'$ also forms the graph of a function $V_G \to C$, meaning that for any vertex $v \in V_G$ we can find a unique $c \in \{1, \dots, k\}$ such that $(v, c)$ belongs to $f'$.

4. Existence says $\forall v \in V_G . \exists c \in C . (v, c) \in f'$, but by Transfer it suffices to prove $\forall^{st} v \in V_G . \exists^{st} c \in C . (v, c) \in f'$. So pick any standard $v \in V_G$. We have $v \in H$ since $H$ contains every standard vertex. The fact that $f$ is a function immediately

gives us a $c \in \{1, \ldots, k\}$ such that $(v, c) \in f$. But $C$ forms a standard finite set, so we can use Theorem 1.2.5 to conclude the standardness of $c$. Now, we have a standard pair $(v, c) \in f$. The Standardization axiom says that $(v, c) \in f'$ holds for standard $v, c$ precisely if $(v, c) \in f$. Hence $(v, c) \in f'$ holds, proving existence.

5. For uniqueness, it suffices to prove that

$$\forall^{st} v \in V_G . \forall^{st} c_1, c_2 \in C . (v, c_1) \in f' \wedge (v, c_2) \in f' \to c_1 = c_2 .$$

So take standard $v, c_1, c_2$ and assume both $(v, c_1) \in f'$ and $(v, c_2) \in f'$. Using the standardness of the pairs $(v, c_1), (v, c_2)$ we can apply Standardization to conclude $(v, c_1) \in f$ and $(v, c_2) \in f$. At that point, we can use the fact that $f$ is a functional relation to conclude $c_1 = c_2$, proving uniqueness.

6. Now we verify that $f'$ gives a $C$-coloring. The sentence

$$\forall v_1, v_2 \in V_G . (v_1, v_2) \in E_G \to f'(v_1) \neq f'(v_2)$$

states this. Transfer applies, so we can get away with showing

$$\forall^{st} v_1, v_2 \in V_G . (v_1, v_2) \in E_G \to f'(v_1) \neq f'(v_2) .$$

So take standard vertices $v_1, v_2$ of the graph and suppose they have an edge between them. Then

$$f'\left(v_1\right) = f\left(v_1\right) \neq f\left(v_2\right) = f'\left(v_2\right)$$

holds: the equalities follow by Standardization, the inequality follows since $f$ is a $C$-coloring. As $v_1, v_2$ get different colors, we conclude that $f'$ gives a $C$-coloring of $G$.

We have concluded that for standard $G$ and $C$, if every finite subgraph of $G$ admits a $C$-coloring, so does the entire graph $G$. However, the conclusion is internal (with standard parameters $G$, $C$), so Transfer makes the standardness assumptions redundant. We get that if every finite subgraph of some graph $G$ admits a $C$-coloring, then the entire graph $G$ admits a $C$-coloring.
**Qed.**

**1.2.8.** In the remainder of this section, we establish some basic results concerning standardness.

**1.2.9. Proposition.** Consider an internal formula $\varphi(x)$ that has one free variable and that does not contain any non-standard parameters. If the sentence $\exists! x.\varphi(x)$ holds, then the object $x$ such that $\varphi(x)$ holds is necessarily standard.

**Proof.** Apply Transfer to $\exists x.\varphi(x)$ to conclude the existence of some standard $x$ satisfying $\varphi(x)$. By the uniqueness clause, every object $y$ satisfying $\varphi(y)$ equals $x$: since $x$ is standard, so is $y$.
**Qed.**

**1.2.10. Corollary.** Given standard sets $A, B$, the sets $A \times B, A \cap B, A \cup B, A \setminus B$ and $\mathcal{P}(B)$ are all standard, and so is the set of all functions $A \to B$.

**Proof.** We can characterize each of them via an internal formula with no non-standard variables (exercise!), and prove their unique existence.
**Qed.**

**1.2.11. Corollary.** Given a standard function $f : A \to B$ and standard $x \in A$, the value $f(x)$ is standard.

**Proof.** Regard the (graph of the) function as a subset of $A \times B$. For each $x \in A$, the internal formula $\varphi(y) \leftrightarrow (x, y) \in f$ contains no non-standard parameters, and it characterizes the value $y = f(x)$ uniquely.
**Qed.**

**1.2.12. Corollary.** One cannot construct a set that contains precisely the standard elements of $\mathbb{N}$.

**Proof.** Assume for a contradiction that we have found such a set $\mathbb{N}^{st}$. All elements of $\mathbb{N}^{st}$ are standard, so by Theorem 1.2.5 $\mathbb{N}^{st}$ is finite. Consider the maximum element $N$ of $\mathbb{N}^{st}$. We have $\mathrm{st}(N)$, so by Corollary 1.2.11, $\mathrm{st}(N+1)$ holds as well. But $N+1 \notin \mathbb{N}^{st}$, a contradiction.
**Qed.**

**1.2.13.** Zermelo-Fraenkel Set Theory proves the (universal closure of the) axiom of induction for any formula $\varphi$ in its language. This internal induction principle remains valid in Internal Set Theory. However, unlike ZFC (but similarly to the theory **PAKSR**

considered in 1.1.10 and in 1.1.22), Internal Set Theory does not prove the axiom of induction for some formulae in its (extended) language. In particular, for $\mathrm{st}(x)$ we have both $\mathrm{st}(0)$ and $\forall x \in \mathbb{N}.\mathrm{st}(x) \to \mathrm{st}(x+1)$ (this follows from Corollary 1.2.11 applied to the standard function $n \mapsto n+1$), but of course not $\forall n \in \mathbb{N}.\mathrm{st}(n)$, which would immediately contradict Idealization. One must maintain constant vigilance not to apply induction arguments to general formulae in the language of Internal Set Theory, especially since we make heavy use of binary predicates in the language of IST in the later chapters of this thesis. On these predicates, we have weaker reasoning principles (among them External Induction, Theorem 1.2.15) available. We develop these below.

**1.2.14. Proposition.** Consider a standard natural number $b$. All $n \in \mathbb{N}$ with $n < b$ are standard.

**Proof.** The internal formula $x < b$ does not have non-standard parameters, so we can construct the finite set $B = \{x \in \mathbb{N} \mid x < b\}$. The standardness of $B$ follows immediately from Proposition 1.2.9, so we can use Theorem 1.2.5 to conclude that all elements of $B$ are standard.
**Qed.**

**1.2.15. Theorem** (External Induction)**.** Take any formula $\varphi$ in the language of Internal Set Theory (possibly with non-standard parameters). If $\varphi(0)$ and $\forall^{st} n.\varphi(n) \to \varphi(n+1)$ both hold, then $\varphi(x)$ holds for all standard $x \in \mathbb{N}$.

**Proof.** Use the axiom of Standardization to construct the set $P = \{\!\!\{x \in \mathbb{N} \mid \varphi(x)\}\!\!\}$. The formula $\psi(x) \leftrightarrow x \in P$ is internal and its only parameter $P$ is standard. Notice that for standard elements $x$, we have $\varphi(x) \leftrightarrow \psi(x)$, so $\psi(0)$ and $\forall^{st} n.\psi(n) \to \psi(n+1)$ both hold, and Transfer applies to the latter, so $\forall n.\psi(n) \to \psi(n+1)$ holds as well. By the (ordinary, internal) induction principle we get $\forall x.\psi(x)$. The desired conclusion, $\forall^{st} x.\varphi(x)$, follows immediately from the equivalence of $\varphi$ and $\psi$ over standard elements.
**Qed.**

## 1.3 Topology via predicates

**1.3.1.** Many applications of Internal Set Theory stem from its ability to transport definitions and techniques meant for finite spaces to the general topological setting. In what follows, we consider topological spaces $(T, \Omega T)$ where $\Omega T$ denotes the lattice of open sets, and $T$ denotes the carrier (underlying set of points). We employ metonymy, and write $T$ instead of $(T, \Omega T)$ whenever we deem the identity of $\Omega T$ sufficiently unambiguous.

**1.3.2.** Given any topological space, we can construct an order relation (often referred to as the *specialization order*) on its points that every continuous function preserves. For finite topological spaces, we can easily achieve the converse as well (Theorem 1.3.6).

**1.3.3. Definition.** Consider a topological space $(T, \Omega T)$, and regard two points $x, y \in T$. We write $x \leq_T y$ if $\forall V \in \Omega T. x \in V \to y \in V$. We call the resulting preorder relation $\leq_T$ the *specialization order* of $T$.

**1.3.4. Exercise.** Check that the construction of Definition 1.3.3 results in a preorder relation on any topological space. Prove that the resulting relation satisfies the partial order axioms precisely on $T_0$-separable spaces.

**1.3.5. Proposition.** Consider a finite topological space $(T, \Omega T)$. A subset $S \subseteq T$ belongs to $\Omega T$ precisely if for each $x, y \in T$ with $x \leq y$, $x \in S \to y \in S$.

**Proof.** Assume that $S \in \Omega T$ and $x \leq y$. Since $x \in S$, the set $S$ forms a neighborhood of $x$, so $S$ contains $y$. That settles one direction. For the other direction, assume $\forall x. \forall y. x \leq y \wedge x \in S \to y \in S$. To prove that $S$ is open, it suffices to show that $S$ contains an open neighborhood of each $x \in S$. But by our assumption it contains at least the open set $\bigcap \{V \in \Omega T \mid x \in V\}$.
**Qed.**

**1.3.6. Theorem** (Birkhoff's representation)**.** Take two finite topological spaces $S$ and $T$. A function $f : S \to T$ is continuous precisely if $x \leq_S y \to f(x) \leq_T f(y)$ holds for all $x, y \in S$.

**Proof.** The comprehension $V_x = \left\{ y \in S \,\middle|\, x \leq_S y \right\}$ constructs the smallest open set containing $x$ for every point $x \in S$ of a finite space $S$. Hence, we only need to verify the openness of preimages of open sets of the form $V_x$ with $x \in T$.
First consider a monotone[4] function $f : S \to T$ and any set of the form $V_x$ for $x \in T$. Assume $a \in f^{-1}(V_x)$ and $a \leq_S b$. We need to prove that $b \in f^{-1}(V_x)$. But $a \in f^{-1}(V_x)$ holds precisely if $x \leq_T f(a)$. Moreover, $f(a) \leq_T f(b)$ follows from $a \leq_S b$ by the monotonicity assumption. We get $x \leq_T f(a) \leq_T f(b)$, and thus $b \in f^{-1}(V_x)$. Since we chose $a, b \in S$ arbitrarily, this proves the continuity of the function $f$.

[4]From here on, monotone functions are always monotone *increasing*.

Now consider a continuous function $f : S \to T$. Assume $a \leq_S b$. By the openness of the preimage of $V_{f(a)}$, we get that $f(a) \leq_T f(a)$ and $a \leq_S b$ together imply $f(a) \leq_T f(b)$. All these preconditions hold, so we can conclude $f(a) \leq_T f(b)$. Since we chose $a, b$ arbitrarily, this proves the monotonicity of the function $f$.
**Qed.**

**1.3.7. Definition.** We call the topological space $(T, \Omega T)$ with underlying set $T = \{\bot, \top\}$ and open set lattice $\Omega T = \{\emptyset, \{\top\}, \{\bot, \top\}\}$ the *Sierpinski space*, and denote it $\acute{S}$.

**1.3.8. Proposition.** Consider any finite topological space $(X, \Omega X)$. We have a one-to-one correspondence between monotone functions $f : X \to \acute{S}$ and open sets $F$ of the space $X$.

**Proof.** Apply Propositions 1.3.5 and 1.3.6.
**Qed.**

**1.3.9.** Alas, we cannot expect analogues of Proposition 1.3.5 and Theorem 1.3.6 to hold for general infinite spaces. For example, applying Definition 1.3.3 to the usual Euclidean topology on the real line $\mathbb{R}$ yields a discrete order, and the same happens on every space with sufficient separation (Proposition 1.3.10), showing cannot reduce the study of topological spaces and continuous functions to the study of functions preserving relations.

**1.3.10. Proposition.** Consider a $\mathrm{T}_1$-separable topological space $(T, \Omega T)$ on which every function that preserves the specialization order $\leq_T$ is continuous. Then $T$ carries the discrete topology.

**Proof.** We show the triviality of the ordering. Consider any two $x, y \in T$ with $x \neq y$. By $\mathrm{T}_1$-separation, we obtain an open set $N$ such that $x \in N$ but $y \notin N$, thus proving $x \not\leq_T y$. Similarly, we get $y \not\leq_T x$. This shows that every function $f : T \to T$ preserves the specialization order, and thus is continuous. In particular, continuity holds for the "characteristic function" mapping elements of $V$ to $x$ and everything else to $y$ for any set $V \subseteq T$. Taking the preimage of $N$ with respect to this characteristic function shows the openness of any set $V$. Hence, $T$ carries the discrete topology.
**Qed.**

## Predicated Spaces and S-continuity

**1.3.11.** Birkhoff's representation theorem requires that the intersection of arbitrary families of open sets itself constitute an open set[5]. This latter property fails badly for general infinite spaces: as we have seen in Proposition 1.3.10, the $\leq_X$ relation gives rise to equality over any $T_1$ space $(X, \Omega X)$, so we have no hope at all of recovering the topology from the specialization relation.

**1.3.12.** In the language of ordinary Zermelo-Fraenkel set theory, every bounded binary predicate $\varphi$ corresponds to a relation (set of ordered pairs), by defining $R$ as $\{(x,y) \in A \times B \mid \varphi(x,y)\}$ where $A, B$ denote the bounds of the predicate $\varphi$. One can't say the same about Internal Set Theory: in the language of IST, we can easily construct predicates that do not form relations. For example, if the predicate $\varphi(x,y)$ abbreviating $x \in \mathbb{N} \wedge y \in \mathbb{N} \wedge \mathrm{st}(x)$ would form a relation, we could define the "set of all standard naturals" as $\{x \in \mathbb{N} \mid \varphi(x,x)\}$, contradicting Corollary 1.2.12.

**1.3.13.** One should see the failure of the correspondence between relations and bounded predicates in Internal Set Theory as an opportunity. The impossibility result of Proposition 1.3.10 applies to any relation, and hence to any bounded ZFC-predicate, but fortunately not to arbitrary predicates in the language of Internal Set Theory! Here we show that by replacing the relation in Definition 1.3.3 with a binary predicate, we can obtain well-behaved analogues of Propositions 1.3.5, Theorem 1.3.6 and even Proposition 1.3.8. This allows us to transport definitions and techniques meant for finite (or more generally: Alexandroff) spaces to the general topological setting. We elected to present these results in detail for two reasons: first, to keep the document self-contained, and second because the ideas inherent in the development will recur in later chapters of the thesis where we characterize Alexandroff approximations and sheaves over Alexandroff spaces. While some of the terminology is novel, all results of the current section are well-known and have appeared in the literature in various forms. The reader should consult the General Topology chapter of *Nonstandard Analysis in Practice* [12] for attributions and alternative formulations.

**1.3.14. Definition.** A *predicated space* $(T, \multimap)$ consists of the following data:

- an underlying set (or carrier set) $T$,
- a binary predicate in the language of Internal Set Theory, $\multimap$, referred to as the

[5] Spaces satisfying this property are called *Alexandroff spaces*.

"nearness", "closeness" or "proximity" predicate,

such that $\forall x.\forall y.x \multimap y \rightarrow x \in T \wedge y \in T$ and $\forall x \in T.x \multimap x$. As usual, we elide $\multimap$ and refer to the predicated space $(T, \multimap)$ simply as $T$ whenever the elision causes no ambiguity.

**1.3.15. Definition.** Consider two predicated spaces $(U, \multimap_U)$ and $(T, \multimap_T)$, along with a function $f : U \rightarrow T$. We call this function *S-continuous* if for all standard $x \in U$ and for all $y \in U$ such that $x \multimap_U y$, we have $f(x) \multimap_T f(y)$.

**1.3.16. Definition.** We call a subset $V \subseteq T$ an *S-open set* of the predicated space $(T, \multimap)$ if for all standard $x \in V$, $V$ contains every $y \in T$ such that $x \multimap y$.
We say that the nearness predicate $\multimap$ *represents the topology of* the standard topological space $(T, \Omega T)$ if every standard S-open set of $(T, \multimap)$ forms an open set in $(T, \Omega T)$ and every standard open set of $(T, \Omega T)$ forms an S-open set in $(T, \multimap)$.
We say that the nearness predicate $\multimap$ *universally represents the topology of* the standard topological space $(T, \Omega T)$ if it represents $(T, \Omega T)$ and for any other predicate $\sim$ representing $(T, \Omega T)$, the implication $\forall^{st} x.\forall y.x \sim y \rightarrow x \multimap y$ holds.

**1.3.17.** At this point the reader should carefully contemplate how would one would formalize the clause defining universal representations in Definition 1.3.16. When we say "for any other predicate representing the space", we have to quantify over all predicates, so no single sentence of set theory (ZFC or IST) suffices for defining universality.

**1.3.18. Proposition.** For any standard topological space $(T, \Omega T)$, we can find a nearness predicate $\multimap$ on $T$ universally representing it.

**Proof.** Define the predicate $x \multimap y$ as an abbreviation of $\forall^{st} N \in \Omega T.x \in N \rightarrow y \in N$. First take a standard open set $S \subseteq T$ of the topological space $(T, \Omega T)$. Consider a pair $x \multimap y$ with $x \in S$ standard. Since $S$ contains a neighborhood of $x$, Transfer assures us that it also contains a standard such neighborhood. That standard neighborhood contains $y$ by definition of the nearness predicate. Thus, $S$ forms an S-open set of $T$.
Now, take a standard S-open set $S \subseteq T$ of the predicated space $(T, \multimap)$. We have that for each standard $x \in T$ and arbitrary $y \in T$ with $x \multimap y$, $x \in S \rightarrow y \in S$. For every finite set of topological neighborhoods of $x$, we can find an open neighborhood of $x$ in $\Omega T$ that forms a subset of all of them (this is a restatement of the fact that the finite intersection of topologically open sets is open). By Idealization we deduce the existence of an open neighborhood of $x$, $I_x \in \Omega T$, that forms a subset of every standard neighborhood of

$x$. By our assumption $S$ contains every $y \in I_x$, so $I_x \subseteq S$. Therefore, $S$ contains a neighborhood $I_x$ of each of standard point $x \in S$. Transfer applies to this statement (since $S$ is standard), and yields that $S$ contains a neighborhood of each of its points, i.e. $S$ is open.

For universality, consider any other predicate $\sim_T$ representing the topology of $T$. We prove the implication $x \sim_T y \to x \multimapinv_T y$ for standard $x \in T$ and arbitrary $y \in T$. Since $\sim_T$ represents the topology of $T$, we have the implication

$$\forall^{st} N.\forall^{st} x \in T.\forall y \in T.(x \sim_T y \wedge N \in \Omega T \wedge x \in N) \to y \in N,$$

and by exchanging connectives and quantifiers we get

$$\forall^{st} x \in T.\forall y \in T.x \sim_T y \to (\forall^{st} N \in \Omega T.x \in N \to y \in N),$$

i.e. $x \sim_T y \to x \multimapinv_T y$, as desired.

**Qed.**

**1.3.19. Theorem.** Take standard topological spaces $(S, \Omega S)$ and $(T, \Omega T)$ universally represented by the nearness relations $\multimapinv_S$ and $\multimapinv_T$ respectively. A standard function $f : S \to T$ forms a continuous map from $(S, \Omega S)$ to $(T, \Omega T)$ precisely if it forms an S-continuous map from $(S, \multimapinv_S)$ to $(T, \multimapinv_T)$.

**Proof.** For the predicates $\multimapinv_S$ and $\multimapinv_T$ constructed in the proof of Proposition 1.3.18 we can simply imitate the proof of Birkhoff's representation theorem (exercise, but you may wish to consult [12]-Section 6.2 for hints). Now consider any other predicates $\sim_S$ and $\sim_T$ representing their respective topologies universally.

Take a standard continuous $f : S \to T$, a standard $x \in S$ and an arbitrary $y \in S$ such that $x \sim_S y$. By the universality of $\multimapinv_S$, the implication $\forall^{st} x'.\forall y'.x' \sim_S y' \to x' \multimapinv_S y'$ holds, so we get $x \multimapinv_S y$. But then $f(x) \multimapinv_T f(y)$ obtains by the proof of the special case. Using the universality of $\sim_T$, we get $f(x) \sim_T f(y)$.

Now take an S-continuous function $f : S \to T$, consider a standard $x \in S$ and an arbitrary $y \in S$ such that $x \multimapinv_S y$. The universality of $\sim_S$ gets us to $x \sim_S y$, so $f(x) \sim_T f(y)$, and the universality of $\multimapinv_T$ immediately yields $f(x) \multimapinv_T f(y)$. Thus, $x \multimapinv_S y$ implies $f(x) \multimapinv_T f(y)$, and by the proof of the special case we obtain the continuity of the function $f$.

**Qed.**

**1.3.20. Exercise.** Consider $\mathbb{R}$ equipped with its usual Euclidean topology, and take a universal representation $\multimap_{\mathbb{R}}$. Construct

1. a continuous map $\mathbb{R} \to \mathbb{R}$ that is not S-continuous;

2. an S-continuous map $\mathbb{R} \to \mathbb{R}$ that is not continuous;

3. a map $\mathbb{R} \to \mathbb{R}$ that is both continuous and S-continuous, but not standard.

**1.3.21.** As we have seen, Theorem 1.3.19 provides an almost perfect counterpart to Birkhoff's representation theorem, and by Proposition 1.3.18, it works for every standard topological space, not only Alexandroff spaces. Thanks to Transfer, we can assume that our topology comes from a binary predicate whenever we want to prove a standard conclusion about an arbitrary topological space. At first sight, the definition of S-continuity (Definition 1.3.15) may look less pleasant than the mere monotonicity of the Alexandroff case, since the former requires an assumption of standardness on the first argument. If one desired an exact correspondence, one could remove this constraint, and *mutatis mutandis* everything would keep working: e.g. one would simply replace the universal representations of Proposition 1.3.18 with $\mathrm{st}(x) \wedge x \multimap y$. However, we fare better by putting up with this minor complication. The payoff comes when we consider functions $f : \mathbb{R} \to \mathbb{R}$ that do preserve the predicate $\multimap_{\mathbb{R}}$ even for non-standard $x$: miraculously, this property corresponds exactly to *uniform continuity*.

**1.3.22. Definition.** Take a predicated space $(T, \multimap)$ on the standard set $T$. We call the structure $(T, \multimap)$ a *topological predicated space* if $\multimap$ universally represents some standard topological space $(T, \Omega T)$.

**1.3.23.** From here on we identify standard topological spaces with topological predicated spaces without any further notice. Thanks to Theorem 1.3.19, we do not need to distinguish between continuous and S-continuous standard functions that go between topological predicated spaces. However, we will rely on nonstandard functions a couple of times in our development: therefore, the reader should expect to see functions for which we explicitly assume both conditions.

## Properties of Predicated Spaces

**1.3.24.** In this subsection we introduce predicated counterparts to the usual properties of topological spaces (separation axioms, compactness, and so on). Customarily, authors attach the "S-" modifier to these generalized concepts (as we did for continuity in Definition 1.3.15), but we will refrain from doing so, reusing names of topological concepts as necessary. The reader should keep in mind that a single property, such as compactness, has infinitely many different generalizations to predicated spaces that all coincide over topological predicated spaces, so there is always some degree of arbitrariness in the choice of the generalization that gets to inherit a particular name.

**1.3.25. Definition.** We call a predicated space $(T, \multimapinv)$

1. *Kolmogorov separable* if two standard points that share exactly the same nearby points are equal;

2. *Fréchet separable* if two nearby standard points are always equal;

3. *Hausdorff separable* if two standard points that share a common nearby point are always equal.

**1.3.26.** It might seem heavy-handed, but the Galactic Halo theorem provides the simplest, most principled way of translating between the common properties of predicated spaces and their topological counterparts. We encourage the readers who skipped Section 1.1.32 to return to that section now and familiarize themselves with at least the proof of Theorem 1.1.39. We assume throughout that the predicate representing a topological space is the one given in the proof of Proposition 1.3.18 (exercise: explain why we do not lose any generality).

**1.3.27. Proposition.** A topological predicated space is Fréchet in the sense of Definition 1.3.25 precisely if it satisfies $T_1$-separation as a topological space.

**Proof.** The following argument implicitly uses the Galactic Halo theorem. We leave it in this form as preparation for the proof of Proposition 1.3.28. Formally, the Fréchet condition states the following:

$$\forall^{st} x, y \in T. x \multimapinv y \to x = y.$$

We expand the definition of $\multimapinv$ (as in Proposition 1.3.18) to get the equivalent condition

$$\forall^{st} x, y \in T. (\forall^{st} N \in \Omega T. x \in N \to y \in N) \to x = y.$$

Putting this in prenex form, we obtain

$$\forall^{st} x, y \in T. \exists^{st} N \in \Omega T. (x \in N \to y \in N) \to x = y.$$

Transfer applies and, we obtain the following Nelson reduction, equivalent to the original condition:

$$\forall x, y \in T. \exists N \in \Omega T. (x \in N \to y \in N) \to x = y.$$

Taking contrapositives gives us the familiar sentence

$$\forall x, y \in T. \exists N \in \Omega T. x \neq y \to x \in N \wedge y \notin N.$$

stating that the space $T$ has $\mathrm{T}_1$-separation.
**Qed.**

**1.3.28. Proposition.** A topological predicated space is Hausdorff in the sense of Definition 1.3.25 precisely if it is $\mathrm{T}_2$-separable (i.e. Hausdorff) as a topological space.

**Proof.** Formally, the Hausdorff condition states the following:

$$\forall^{st} x, y \in T. \forall s \in T. (x \multimapinv s \wedge y \multimapinv s) \to x = y.$$

We start by expanding the definition of $\multimapinv$, and get the equivalent statement

$$\begin{aligned}&\forall^{st} x, y \in T. \forall s \in T.\\ &((\forall^{st} N \in \Omega T. x \in N \to s \in N) \wedge (\forall^{st} M \in \Omega T. y \in M \to s \in M)) \to x = y.\end{aligned}$$

Applying the algorithm of the Galactic Halo theorem, we get the equivalence of the original condition with the following monstrosity:

$$\begin{aligned}&\forall x, y \in T. \exists N', M' \in \mathcal{P}^{\mathrm{fin}}(\Omega T). \forall s \in T. \exists N \in N'. \exists M \in M'.\\ &((x \in N \to s \in N) \wedge (y \in M \to s \in M)) \to x = y.\end{aligned}$$

Replacing $A \to B$ with $\neg A \vee B$ everywhere yields the more legible, equivalent condition

$$\begin{aligned}&\forall x, y \in T. x \neq y \to \exists N', M' \in \mathcal{P}^{\mathrm{fin}}(\Omega T). \forall s \in T. \exists N \in N'. \exists M \in M'.\\ &(x \in N \wedge s \notin N) \vee (y \in M \wedge s \notin M).\end{aligned}$$

Since taking the union $\bigcup N'$ results in an open set of $T$ (and similarly for $M'$), we get a much simpler equivalent condition,

$$\forall x, y \in T. x \neq y \to \exists N, M \in \Omega T. \forall s \in T. (x \in N \wedge s \notin N) \vee (y \in M \wedge s \notin M).$$

Setting $s = x$ in the above condition proves $y \in M$ and $x \notin M$, while setting $s = y$ proves $x \in N$ and $y \notin N$. Thus we get that $N$ $(M)$ forms a neighborhood of $x$ (resp. $y$), and

$$\forall x, y \in T. x \neq y \to \exists N, M \in \Omega T. \forall s \in T. s \notin N \vee s \notin M,$$

proving $N \cap M = \emptyset$, proving $T_2$-separation for $T$. Clearly, the converses of the last two implications obtain as well, so a topological predicated space is Hausdorff in the sense of Definition 1.3.25 precisely if it has $T_2$-separation.
**Qed.**

**1.3.29. Proposition.** A topological predicated space satisfies the Kolmogorov condition of Definition 1.3.25 precisely if it satisfies $T_0$-separation.

**Proof.** Consider a standard $T_0$ space $T$ and two standard points $x, y \in T$. Assume $\forall s \in T. x \multimap s \leftrightarrow y \multimap s$. We have $x \multimap x$ and $y \multimap y$ by reflexivity, and setting $s = y$ in our assumption gives $x \multimap y$. Setting $s = x$ yields $y \multimap x$. Now assume for a contradiction the existence of an open set $N$ containing $x$ but not $y$. By Transfer we'd have a standard such $N$, and since $x \multimap y$, we'd have $y \in N$, a contradiction. Hence, such an $N$ cannot exist. We get the same conclusion if we assume the existence of $M$ containing $y$ but not $x$. From $T_0$-separation it follows that $x = y$.
Now consider a Kolmogorov topological predicated space $T$, and take two of its points, $x, y \in T$. We can provisionally assume the standardness of both $x$ and $y$. Assume that every open set $N$ that contains $x$ also contains $y$, and vice versa. A fortiori the same holds for all standard sets. Hence for all $s \in T$ such that $x \multimap s$, we have that a standard neighborhood of $y$ contains $x$, and hence contains $s$, so $y \multimap s$. Similarly if we switch $x$ and $y$. Thus, $\forall s. x \multimap s \leftrightarrow y \multimap s$, and by the Kolmogorov condition we conclude $x = y$. Thus, if for two standard points $x, y$ we cannot find an open set $N$ containing one but not the other, then $x = y$. Given the internality of our conclusion, we can discharge the provisional assumptions of standardness, which proves $T_0$-separation for the space $T$.
**Qed.**

**1.3.30. Proposition.** A predicated space that satisfies the Hausdorff separation property also satisfies the Fréchet separation property. A predicated space that satisfies the Fréchet separation property also satisfies the Kolmogorov separation property.

**Proof.** Assume that the predicated space $(T, \multimapinv)$ satisfies Hausdorff separation. Then we have

$$\forall^{st} x, y \in T. \forall s \in T. (x \multimapinv s \wedge y \multimapinv s) \to x = y.$$

Setting $s = x$, and using the fact that $x \multimapinv x$ holds by reflexivity, we obtain

$$\forall^{st} x, y \in T. y \multimapinv x \to x = y,$$

so $(T, \multimapinv)$ satisfies Fréchet separation. Now, consider standard $x, y \in T$ such that $\forall s. x \multimapinv s \leftrightarrow y \multimapinv s$. Set $s = x$, and use reflexivity to conclude $y \multimapinv x$. By the Fréchet condition $x = y$, so $(T, \multimapinv)$ satisfies Kolmogorov separation.
**Qed.**

**1.3.31.** Notice that we did not need to restrict Proposition 1.3.30 to topological predicated spaces: the conclusion holds on any predicated space, regardless of the standardness of the carrier. The reverse implications do not hold: take your favorite standard non-Hausdorff $\mathrm{T}_1$-space and a non-$\mathrm{T}_1$ $\mathrm{T}_0$-space as counterexamples.

**1.3.32. Definition.** We call a predicated space *compact* if every point of the space lies near a standard point of the space.

**1.3.33. Theorem** (Robinson's characterization)**.** A topological predicated space $(T, \multimapinv)$ satisfies Definition 1.3.32 (compactness) precisely if every open cover in $(T, \Omega T)$ has a finite subcover.

**Proof.** Formally, the compactness condition states the following:

$$\forall y \in T. \exists^{st} x \in T. x \multimapinv y.$$

We start by expanding the definition of $\multimapinv$, and get the equivalent statement

$$\forall y \in T. \exists^{st} x \in T. \forall^{st} M \in \Omega T. x \in M \to y \in M. \qquad (\star)$$

We wish to apply the Galactic Halo theorem to $(\star)$. To accomplish that, recall that the formula

$$\exists^{st} x \in T . \forall^{st} M \in \Omega T . x \in M \to y \in M$$

would have the following Nelson normal form:

$$\forall^{st} N : T \to \Omega T . \exists^{st} x \in T . x \in N(x) \to y \in N(x).$$

Therefore, applying the Galactic Halo theorem outputs the equivalent condition

$$\forall N : T \to \Omega T . \exists X \in \mathcal{P}^{\mathrm{fin}}(T) . \forall y \in T . \exists x \in X . x \in N(x) \to y \in N(x)$$

when applied to the sentence ($\star$). It suffices to prove this condition equivalent to "every open cover having a finite subcover".

First assume that the condition holds and consider an open cover $U \subseteq \Omega T$ of the space $T$. By the Axiom of Choice we can get a function $N : T \to \Omega T$ that assigns to each point $x \in T$ a covering set $U_x \in U$ such that $x \in U_x$. By the assumed condition, we have a finite set of points $X \subseteq T$ such that $\forall y \in T . \exists x \in X . y \in U_x$. Thus, the set $V = \left\{ S \in \Omega T \mid \exists x \in X . S = U_x \right\}$ constitutes a finite subcover of $U$.

Now assume that every open cover has a finite subcover. Consider any function $N$ from $T$ to $\Omega T$. If we can find a point $q \in T$ such that $q \notin N(q)$, then we can set $X = \{q\}$, and $\forall y \in T . q \in N(q) \to y \in N(q)$ holds vacuously. If we cannot find such a point $q$, then $N$ constitutes an open cover of $T$, and its finite subcover gives rise to the desired set of points $X$.

**Qed.**

**1.3.34. Definition.** We call a predicated space an *equivalence space* if its nearness predicate satisfies transitivity and symmetry.

**1.3.35. Proposition.** Every metric space admits a universal representation as an equivalence space.

**Proof.** Consider any metric space $M$ equipped with a metric $d : M \to \mathbb{R}$. Define the predicate $x \approx y$ as an abbreviation for $\forall^{st} \varepsilon > 0 . d(x, y) < \epsilon$. The reflexivity of $\approx$ follows from $\forall x, y \in M . d(x, y) = 0 \leftrightarrow x = y$. The symmetry of $\approx$ follows directly from the fact that $\forall x, y \in M . d(x, y) = d(y, x)$. To prove transitivity, consider $x, y, z \in M$ such that $x \approx y$ and $y \approx z$. Take any standard $\varepsilon > 0$. Since $\frac{\varepsilon}{4}$ is standard by Proposition 1.2.9, we have both $d(x, y) < \frac{\varepsilon}{4}$ and $d(y, z) < \frac{\varepsilon}{4}$. The triangle inequality gives $d(x, z) \leq d(x, y) +$

$d(y,z) \le \frac{\varepsilon}{2} < \varepsilon$. Since $d(x,z) < \varepsilon$ for any standard $\varepsilon$, we get $x \approx z$ as required.
Now we must prove that $\approx$ represents the metric topology carried by $M$. First consider a standard open set $V$ of the topological space $M$, pick any standard $x \in V$ and consider a nearby point $y \approx x$. By the openness of $V$ in the metric topology, we can find an open ball $B$ containing $x$ with $B \subseteq V$. By considering the radius $r$ of $B$, we get that $\exists r > 0.\forall y.d(x,y) < r \to y \in V$. Transfer applies, so $\exists^{st} r > 0.\forall y \in M.d(x,y) < r \to y \in V$. Using $y \approx x$ and the standardness of $r$ we have $d(x,y) < r$. Hence, $y \in V$.
Now consider a standard set $V \subseteq M$ such that $\forall^{st} x \in M.\forall y \in M.x \approx y \wedge x \in V \to y \in V$. Expanding the definition of $\approx$ and putting the resulting statement in prenex form gives

$$\forall^{st} x \in M.\forall y \in M.\exists^{st} \varepsilon > 0.d(x,y) < \varepsilon \wedge x \in V \to y \in V.$$

Since the prenex form has no non-standard parameters, we can apply the Galactic Halo theorem to obtain the following equivalent condition:

$$\forall x \in M.\exists E \in \mathcal{P}^{\text{fin}}(\mathbb{R}_+).\forall y \in M.\exists \varepsilon \in E.d(x,y) < \varepsilon \wedge x \in V \to y \in V.$$

Taking the minimum of $E$, we get the further equivalent condition

$$\forall x \in M.\exists \varepsilon > 0.\forall y \in M.d(x,y) < \varepsilon \wedge x \in V \to y \in V$$

which implies the more legible condition

$$\forall x \in V.\exists \varepsilon > 0.\forall y \in M.d(x,y) < \varepsilon \to y \in V.$$

But then $V$ contains a ball of radius $r = \varepsilon$ around each of its points $x \in V$, and consequently $V$ is open in the metric topology of $M$.
Finally, we need to show the universality of $\approx$. By the universality of the predicate $\multimap$ of Proposition 1.3.18, it suffices to prove $\forall^{st} x.\forall y.x \multimap y \to x \approx y$. So assume $x \multimap y$ and take any standard $\varepsilon > 0$. The open ball $B$ of radius $\varepsilon$ around $x$ forms an open set of the metric topology, and is standard by Proposition 1.2.9. From $x \multimap y$, $\mathrm{st}(B)$ and $x \in B$ we have $y \in B$. But then $d(x,y) < \varepsilon$, and since $\varepsilon$ was arbitrary, we get $x \approx y$.
**Qed.**

## Ultrafilters

**1.3.36.** We often rely on the following well-known results about ultrafilters in the subsequent chapters, especially in results about structural approximation. The proof strategy consists of little more than making the observation that ultrafilters correspond to types (in the sense of model theory) of non-standard elements. The only novelty of the section occurs in the (frankly, totally unsurprising) characterization of metric ultraproducts in Proposition 1.3.43. The reader may wish to consult the article *Ultrafilters and ultraproducts in non-standard analysis* [8] by Cherlin and Hirschfeld for a discussion of the same subject from a model-theoretic perspective.

**1.3.37. Definition.** An ultrafilter $\mathcal{F}$ over some set $I$ *has a monadic element* if we can find some $\omega \in I$ that belongs to every standard element of $\mathcal{F}$.

**1.3.38. Proposition.** Every ultrafilter $\mathcal{F}$ over some set $I$ has a monadic element.

**Proof.** Consider any standard finite non-empty subset $\mathcal{S}$ of $\mathcal{F}$. By the finite intersection property, $\bigcap \mathcal{S} \in \mathcal{F}$. Since $\emptyset \notin \mathcal{F}$, we can find some $x \in \bigcap \mathcal{S}$, and of course that $x$ satisfies $\forall S \in \bigcap \mathcal{S}.x \in S$. Given the internality of this conclusion, we can apply Idealization and get a single $x \in I$ that belongs to all standard sets $S \in \mathcal{F}$.
**Qed.**

**1.3.39. Proposition.** A standard ultrafilter is non-principal precisely if it has a non-standard monadic element.

**Proof.** A standard principal ultrafilter $\mathcal{F}$ has a unique standard generator $x$ by Proposition 1.2.9, and $x$ belongs to every element of $\mathcal{F}$, so *a fortiori* it constitutes a standard monadic element of $\mathcal{F}$. In fact, $\mathcal{F}$ has a unique monadic element in this case, since the singleton set $\{x\}$ forms a standard element of $\mathcal{F}$, so by definition every monadic element belongs to the set $\{x\}$.
Assume that the standard non-principal ultrafilter $\mathcal{F}$ has a standard monadic element $x \in I$. Then $\forall^{st} S \in \mathcal{F}.x \in S$ holds. This formula contains no non-standard parameters, so Transfer applies and we can conclude $\forall S \in \mathcal{F}.x \in S$, contradicting the non-principality of $\mathcal{F}$.
**Qed.**

**1.3.40. Lemma** (Ultrafilter)**.** Every infinite set $I$ admits a non-principal ultrafilter.

**Proof.** Provisionally assume the standardness of $I$. By Theorem 1.2.5, the infinite set $I$ contains some non-standard element $\omega \in I$. Consider the standard set

$$\mathfrak{F} = \{\!\{ S \in \mathcal{P}(I) \mid \omega \in S \}\!\}$$

defined using the Standardization axiom. The set $\mathfrak{F}$ forms an ultrafilter. We prove that $\mathfrak{F}$ satisfies the finite intersection property, but leave the other properties as exercises for the reader. Take standard $A, B \in \mathfrak{F}$. By the defining property of $\mathfrak{F}$, we get $\omega \in A$ and $\omega \in B$, so $\omega \in A \cap B$. Hence, the implication $A \in \mathfrak{F} \wedge B \in \mathfrak{F} \rightarrow A \cap B \in \mathfrak{F}$ holds for standard $A, B$, and by Transfer for every $A, B$.
The ultrafilter $\mathfrak{F}$ has $\omega$ as a monadic element since $\omega \in S$ holds for every standard $S \in \mathfrak{F}$ by definition of the set $\mathfrak{F}$. But we started by choosing a non-standard $\omega \in I$, so an application of Proposition 1.3.39 shows that $\mathfrak{F}$ is non-principal. We have proved that every standard infinite set admits a non-principal ultrafilter. Hence, by Transfer, every infinite set admits a non-principal ultrafilter.
**Qed.**

**1.3.41.** Notice that we made no use of the Axiom of Choice in the proof of Lemma 1.3.40. Since Zermelo-Fraenkel Set Theory without Choice does not prove the Ultrafilter Lemma, we have now established our claim in 1.1.46 that Internal Set Theory with the Axiom of Choice removed does not extend Zermelo-Fraenkel Set Theory conservatively.

**1.3.42. Theorem.** Consider a standard index set $I$, a standard $I$-indexed family of sets $A$ and a standard ultrafilter $\mathcal{F}$ on the set $I$. Let $\omega \in I$ denote a monadic element of $\mathcal{F}$. Take two standard elements $[f], [g]$ of the ultraproduct $\prod_{i \in I} A_i / \mathcal{F}$. We have $[f] = [g]$ precisely if for any standard $f \in [f], g \in [g]$ we have $f(\omega) = g(\omega)$.

**Proof.** The equality $[f] = [g]$ holds precisely if the set $\{i \in I \,|\, f(i) = g(i)\}$ belongs to the ultrafilter $\mathcal{F}$ for some (indeed, any) representatives $f \in [f]$ and $g \in [g]$. Using the standardness of $f, g, I$, we can conclude the standardness of the set $\{i \in I \,|\, f(i) = g(i)\}$. The monadic element $\omega$ must therefore belong to this set, giving $f(\omega) = g(\omega)$ as required. The other direction works identically.
**Qed.**

**1.3.43. Theorem.** Consider a standard index set $I$, a standard number $k \in \mathbb{R}$ a standard $I$-indexed family of groups $G$, each $G_i$ equipped with a standard bi-invariant $k$-bounded metric $d_i$, and pick a standard ultrafilter $\mathcal{F}$ on the set $I$. Take two standard elements

$[f],[g]$ of the the metric ultraproduct group $\prod_{y\in I} A_i/d_{\mathcal{F}}$. We have $[f]=[g]$ precisely if for any standard $f\in[f], g\in[g]$ we have $\forall^{st}\varepsilon>0. d_\omega(f(\omega),g(\omega))<\varepsilon$ where $\omega$ denotes a monadic element of $\mathcal{F}$.

**Proof.** By the definition of metric ultraproducts ([13]-Definition 2.1.) we have $[f]=[g]$ in $\prod_{y\in I} A_i/d_{\mathcal{F}}$ precisely if for any $\varepsilon>0$ the set $\left\{i\in I \,\middle|\, d_i(f(i),g(i))<\varepsilon\right\}$ belongs to the ultrafilter $\mathcal{F}$. So take standard $f,g,\varepsilon$. Then we have $\operatorname{st}\left(\left\{i\in I \,\middle|\, d_i(f(i),g(i))<\varepsilon\right\}\right)$; by Definition 1.3.37 we get $\omega\in\left\{i\in I \,\middle|\, d_i(f(i),g(i))<\varepsilon\right\}$, and consequently we must have $d_\omega(f(\omega),g(\omega))<\varepsilon$. For the other direction assume that $\forall^{st}\varepsilon>0. d_\omega(f(\omega),g(\omega))<\varepsilon$ holds. Reversing the previous argument, we have that for standard $\varepsilon$ the set of indices $\left\{i\in I \,\middle|\, d_i(f(i),g(i))<\varepsilon\right\}$ belongs to $\mathcal{F}$ as required. Transfer gives the full result.
**Qed.**

## Brouwer's fixed point theorem

**1.3.44.** To finish off this chapter, and to illustrate the use of the tools introduced in the previous sections, we present an Internal Set Theory proof of Brouwer's fixed point theorem, similar to (but simpler than) the standard combinatorial proof going through Sperner's coloring lemma.

**1.3.45. Theorem** (Brouwer's fixed point)**.** Every continuous function mapping the unit disk $D\subseteq\mathbb{R}^2$ to itself has a fixed point.

**Proof.** Identify $D$ with the unit disk in $\mathbb{C}$ the obvious way. Let $\approx$ denote the universal nearness predicate on $\mathbb{C}$ that comes from the proof of Proposition 1.3.35. Consider any continuous function $f: D\to D$ and provisionally assume the standardness of $f$. Take a finite set $H\subseteq D$ that contains every standard point of $D$ (use Idealization). Consider the labeling function $\ell: D\to\{0,1,2,3\}$ of Figure 1.1, defined by the expression

$$\ell(x)=\begin{cases}0 & \text{if } f(x)-x=0;\\ k+1 & \text{if } f(x)-x\neq 0 \text{ and } \arg(f(x)-x)\in\left[\frac{2}{3}k\pi,\frac{2}{3}(k+1)\pi\right).\end{cases}$$

If we have $x\in H$ such that $\ell(x)=0$, then $f$ has the fixed point $x$. Otherwise, we can break the boundary of the disk into three circular arcs such that on each arc $\ell$ takes exactly two values. Thus, by using Sperner's lemma ([20]-Theorem 2.6) we can find three points $x_1,x_2,x_3\in H$ such that $\ell(x_1)=1,\ell(x_2)=2,\ell(x_3)=3$ and $H$ contains no points that lie inside the triangle formed by the three points. This implies that said triangle contains no standard points, and hence $x_1 \approx x_2 \approx x_3$. Since $D$ is compact, we can use Theorem 1.3.33 to find a standard point $x$ simultaneously near $x_1, x_2$ and $x_3$. This means that the complex number $f(x) - x$ lies infinitesimally close to numbers with arguments in $\left[0, \frac{2}{3}\pi\right)$, $\left[\frac{2}{3}\pi, \frac{4}{3}\pi\right)$ and $\left[\frac{4}{3}\pi, 2\pi\right)$. A moment's thought (or a glance at Figure 1.1) shows that the only standard complex number satisfying such a requirement is zero. Therefore $f(x) - x = 0$, and $x$ constitutes a fixed point for the function $f$.

**Qed.**

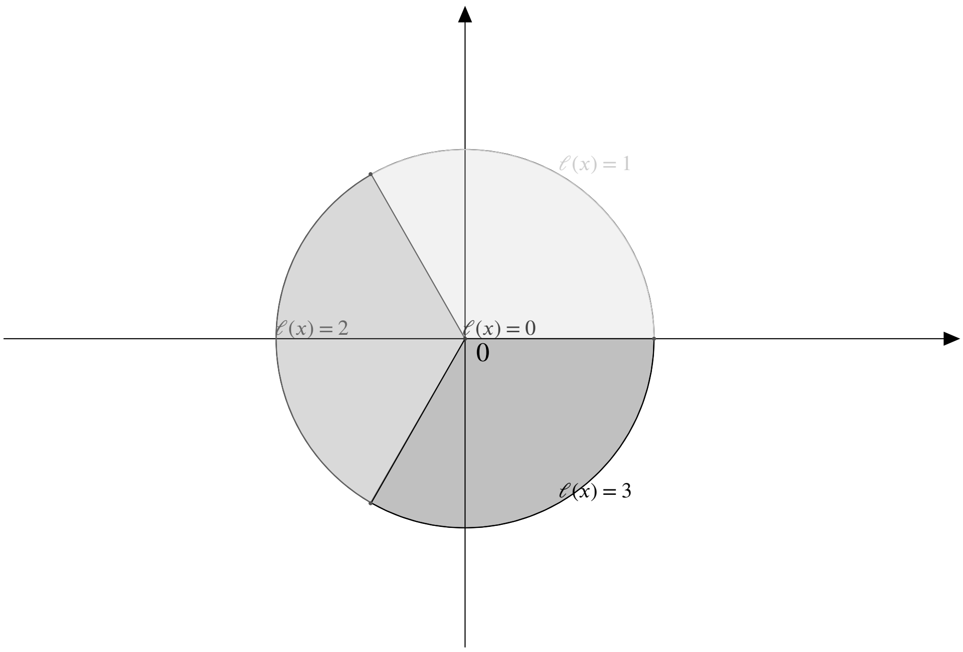


Figure 1.1: Values of the labeling map $\ell$ on the unit circle.

# Chapter 2

# Structural Approximation

## 2.1 Motivation

**2.1.1.** Somewhat orthogonally to Internal Set Theory, developments in Stability Theory led to an idea of dealing with very large finite structures as if they were approximating models of uncountably categorical theories. Zilber [50] introduced such a theory of approximations with an eye towards applications in physics. Since we define a more general form of approximation below, we attach the adjective *ordinary* to Zilber's notion. As in Pillay [38] and Zilber [51]-Section 3, we restrict our attention to the cases where the ordinary approximating object consists of a literal ultraproduct of structures, and not merely an elementary extension of one. This usage has the advantage of being consistent with the more recent applications of the technique in the work of Morales and Zilber [31].

**2.1.2. Definition** ([51]-Definition 3.2)**.** Fix some first-order theory $T$. Consider a model $\mathbf{M}$ of $T$ and an $I$-indexed family $M$ of models of the same theory. An *ordinary structural approximation* of $\mathbf{M}$ consists of the following data:

- an ultrafilter $D \subseteq \mathcal{P}(I)$, and
- a surjective $T$-homomorphism $\lim : \prod_{i \in I} M_i / D \to \mathbf{M}$

where $\prod_{i \in I} M_i / D$ denotes the ultraproduct of $M$ over the ultrafilter $D$. If the codomain of the indexed set $M$ consists exclusively of finite $T$-structures, we speak of an *ordinary finite approximation*.

**2.1.3. Definition** ([50]-Definition 2.5)**.** Fix notation as in Definition 2.1.2. An *ordinary strong approximation* of $\mathbf{M}$ consists of the following data:

- an ultrafilter $D \subseteq \mathcal{P}(I)$,
- a surjective $T$-homomorphism $\lim : \prod_{i \in I} M_i/D \to \mathbf{M}$, and
- a $T$-homomorphism $\operatorname{colim} : \mathbf{M} \to \prod_{i \in I} M_i/D$

such that $\lim(\operatorname{colim} x) = x$ for all $x \in \mathbf{M}$. If the codomain of the indexed set $M$ consists exclusively of finite $T$-structures, we speak of an *ordinary strong finite approximation*.

**2.1.4. Proposition.** The group $\mathbb{Z}_p$ of $p$-adic integers admits an ordinary finite cyclic approximation and the analoguos result holds when we consider $\mathbb{Z}_p$ as a ring. The group $\hat{\mathbb{Z}}$ (the profinite completion of the integers) admits an ordinary strong finite cyclic approximation.

**Proof.** See [51]-Proposition 4.3 and [50]-Proposition 1. These results follow from our own Corollary 2.2.19 as well.
**Qed.**

**2.1.5. Proposition.** The field $\mathbb{C}$ of complex numbers admits an ordinary finite approximation. However, the field $\mathbb{R}$ of real numbers does not admit any such approximation.

**Proof.** See [51]-Proposition 5.2.
**Qed.**

**2.1.6.** Before Zilber introduced strong approximation, Gordon [1] investigated the sense in which the finite Fourier transform can approximate the Fourier transform in the Hilbert space of functions on a locally compact group, which led to the synthesis of the concept of locally embeddably finite (LEF) groups. LEF and strongly approximable groups often coincide, e.g. vector spaces are strongly approximable precisely if LEF.

**2.1.7. Definition** ([6]-Theorem 7.2.5. *sic!*)**.** We call a group $G$ *locally embeddably finite* or *LEF* if we can find an $I$-indexed family of groups $M_i$, ultrafilter $D \subseteq \mathcal{P}(I)$ and injective group homomorphism $\operatorname{colim} : G \hookrightarrow \prod_{i \in I} M_i/D$.

**2.1.8.** Zilber [51] poses the following question: can a sequence of finite groups give an ordinary finite approximation to the group $\mathrm{SO}_3(\mathbb{R})$? Using nonstandard analysis in superstructures, Pillay [38] rephrased the problem in terms of Bohr compactifications, tentatively conjecturing that $bG^0$ is commutative for any pseudo-finite group $G$. Nikolov, Schneider and Thom [34] settled this conjecture in the positive. Their results not only give a negative answer to Zilber's original question, but establish the much stronger result that one cannot approximate *any* compact simple Lie group in the sense of Definition 2.1.2 using finite groups.

**2.1.9.** The results mentioned in 2.1.8 establish a significant gap between groups that have finite approximations and those groups that don't have any such approximation. We prove that profinite groups always admit ordinary finite approximations (Proposition 2.2.20), and our main theorem (Theorem 2.3.9) can be used to obtain finer-grained statements about groups that admit strong approximations by finite groups in fixed, particular forms (we give two examples in Section 2.3.11).

## 2.2 Approximation in IST

**2.2.1.** We start by proposing a new notion of strong approximation in the language of Internal Set Theory that abstracts away from first-order structures. The new notion is strictly more general than Zilber's approximations (although it will take us some time to actually prove this, in Proposition 2.2.32) and encompasses all of the common finitariness conditions in group theory. Indeed, ordinary approximation and the LEF condition both give rise to well-behaved instances of the proposed definition.

**2.2.2. Definition.** Consider a set $H$ and a Fréchet predicated space $(G, \multimap_G)$ with $G$ standard. We call a binary predicate $\imath(x,y)$ where $x$ ranges over elements of $H$ and $y$ ranges over elements of $G$ a *weak approximation* of $(G, \multimap_G)$ via $H$ if it satisfies the following existence-uniqueness conditions:

1. For any standard $g \in G$ we can find $h \in H$ such that $\imath(h,g)$ holds.
2. For any $g_1, g_2 \in G$, if we can find $h \in H$ such that $\imath(h,g_1)$ and $\imath(h,g_2)$ both hold, then $g_1 \multimap_G g_2$.

For finite $H$, we label the weak approximation *finite*. If $(G, 1, \cdot)$ and $(H, 1_H, \cdot_H)$ form groups, and $\imath$ is a *logical relation of groups* in the sense that all of

1. $\imath(1_H, 1)$,
2. $\forall^{st} g \in G. \forall h \in H. \imath(h,g) \to \imath(h^{-1}, g^{-1})$, and
3. $\forall^{st} g_1, g_2 \in G. \forall h_1, h_2 \in H. \imath(h_1, g_1) \wedge \imath(h_2, g_2) \to \imath(h_1 \cdot_H h_2, g_1 \cdot g_2)$

hold, then we say that $H$ *weakly approximates* $G$ *as a group*.

**2.2.3. Definition.** Consider a Fréchet predicated space $(G, \multimap_G)$ with $G$ standard, and an arbitrary set $H$. We call a binary predicate $\iota$ an *approximation* of $(G, \multimap_G)$ via $H$ if

1. the predicate $\iota$ weakly approximates $(G, \multimap_G)$ via $H$, and
2. for each standard $g \in G$, whenever we can find $h_1, h_2 \in H$ with $\iota(h_1, g)$ and $\iota(h_2, g)$, we have $h_1 = h_2$

As in Definition 2.2.2, we speak of a *finite approximation* when $H$ is a finite set. If $(G, \cdot)$ and $(H, \cdot_H)$ form groups, and $\iota$ is a logical relation of groups in the sense that $\forall^{st} g_1, g_2 \in G. \forall h_1, h_2 \in H. \iota(h_1, g_1) \wedge \iota(h_2, g_2) \to \iota(h_1 \cdot_H h_2, g_1 \cdot g_2)$ holds, then we say that $H$ *approximates* $G$ *as a group*.

**2.2.4.** Analogously to group approximations, we can define approximations of other first-order structures (we briefly discuss how to do this in 2.2.12). We wish to relate strong approximation to the construction of the nonstandard finite sets $H$ used in the proofs of Theorems 1.2.7 and 1.3.45. Indeed, as we will see in Proposition 2.2.7, Definition 2.2.3 axiomatizes some obvious properties of the inclusion map of $H$. For this reason we may sometimes opt to use functional notation, or choose to represent $\iota$ as an arrow in diagrams, even when $\iota$ does not stand for a function or functional predicate.

**2.2.5. Definition.** Consider a standard set $G$. Let the binary predicate $g_1 \multimap g_2$ abbreviate the formula $\mathrm{st}(g_1) \wedge \mathrm{st}(g_2) \to g_1 = g_2$. We say that $H$ *approximates* $G$ *as a set* (without mentioning any specific predicate $\multimap_G$ on $G$) when $H$ approximates $(G, \multimap)$.

**2.2.6.** When $H$ approximates a Fréchet space $(G, \multimap)$, it also approximates $G$ as a set.

**2.2.7. Proposition.** Every standard set $G$ admits a finite approximation $H$.

**Proof.** For every standard finite subset $F \in \mathcal{P}^{\mathrm{fin}}(G)$, we can find a finite set $H$ that contains every element of $F$ (trivially, just set $F = H$). The axiom of Idealization applies to this statement, and yields the existence of a single finite set $H \in \mathcal{P}^{\mathrm{fin}}(G)$ that nevertheless contains every standard element of $G$. We can identify the predicate $\iota$ with the graph of the inclusion map $H \hookrightarrow G$. All three required properties hold:

1. For any standard $g \in G$, we have $g \in H$, so $\iota(g) = g$ holds.
2. For any standard $g \in G$ and $h_1, h_2 \in H$ such that $\iota(h_1) = g$ and $\iota(h_2) = g$, we have $h_1 = \iota(h_1) = g = \iota(h_2) = h_2$.

3. For any standard $g_1, g_2 \in G$, and $h \in H$ such that $\imath(h) = g_1$ and $\imath(h) = g_2$, we simply have $g_1 = \imath(h) = g_2$.

**Qed.**

**2.2.8. Proposition.** Consider a standard ordinary strong approximation of a structure $G$ via the $I$-indexed sequence $H$. We can find $\omega \in I$ and an internal binary predicate $\imath$ (relating elements of $H_\omega$ to elements of $G$) that satisfy the following conditions:

1. For any $g \in G$ we can find $h \in H_\omega$ such that $\imath(h, g)$ holds.
2. For any $g \in G$, $h_1, h_2 \in H_\omega$ such that $\imath(h_1, g)$ and $\imath(h_2, g)$ hold, we have $h_1 = h_2$.
3. For any standard $g_1, g_2 \in G$ and any $h \in H_\omega$ such that $\imath(h, g_1)$ and $\imath(h, g_2)$ both hold, we have $g_1 = g_2$.

**Proof.** Denote the $I$-ultrafilter coming from the ordinary approximation as $\mathcal{D}$. Fix a standard right inverse $r : \prod_{i \in I} H_i/\mathcal{D} \to \prod_{i \in I} H_i$ of the quotient map $[-] : \prod_{i \in I} H_i \to \prod_{i \in I} H_i/\mathcal{D}$. By Proposition 1.3.38 the ultrafilter $\mathcal{D}$ has a monadic element $\omega \in I$. Define $\imath(h, g)$ between $H_\omega$ and $G$ as an abbreviation for the formula $(r \circ \mathrm{colim})(g)(\omega) = h$. We verify the three conditions.

1. For any $g \in G$, one can regard the element $(r \circ \mathrm{colim})(g)$ of the Cartesian product of the family $H$ as a function with signature $(i \in I) \to H_i$. Hence we have $(r \circ \mathrm{colim})(g)(\omega) \in H_\omega$, so the first condition holds for $h = (r \circ \mathrm{colim})(g)(\omega)$.
2. Take any $g \in G$ and $h_1, h_2 \in H_\omega$. Assume that $\imath(h_1, g)$ and $\imath(h_2, g)$ both hold. Then $h_1 = (r \circ \mathrm{colim})(g)(\omega) = h_2$ as desired.
3. Take any standard $g_1, g_2 \in G$, and any $h \in H_\omega$. Assume that $\imath(h, g_1)$ and $\imath(h, g_2)$ both hold. Since we chose $r$ as a standard function, and we have st(colim) by the standardness of the approximation, Corollary 1.2.11 guarantees the standardness of the functions $(r \circ \mathrm{colim})(g_1)$ and $(r \circ \mathrm{colim})(g_2)$. By our assumptions we have

   $$(r \circ \mathrm{colim})(g_1)(\omega) = h = (r \circ \mathrm{colim})(g_2)(\omega),$$

   so these functions take the same value at the monadic element $\omega$ of $\mathcal{D}$. Applying Theorem 1.3.42, we immediately obtain

   $$[(r \circ \mathrm{colim})(g_1)] = [(r \circ \mathrm{colim})(g_2)],$$

and since $r$ forms a section of the quotient map $[-] : \prod_{i \in I} H_i \to \prod_{i \in I} H_i/\mathcal{D}$, we are now in the position to deduce $\mathrm{colim}(g_1) = \mathrm{colim}(g_2)$. Applying lim to both sides of the equation yields $g_1 = g_2$ as desired.

**Qed.**

**2.2.9. Corollary.** Assume that we have a standard ordinary approximation of a structure $G$ via the $I$-indexed sequence $H$. Then $H_\omega$ approximates $G$ as a set for some index $\omega \in I$.

**Proof.** Immediate from Proposition 2.2.8.
**Qed.**

**2.2.10. Proposition.** Consider a standard ordinary approximation of a structure $G$ via the $I$-indexed sequence $H$. We can find $\omega \in I$ and a binary predicate $\iota$ (relating elements of $H_\omega$ to elements of $G$) such that $\iota$ weakly approximates the set $G$ via $H_\omega$.

**Proof.** Let $\iota(h, g)$ abbreviate $\exists^{st} f : (i \in I) \to H_i.\lim[f] = g \wedge f(\omega) = h$. The required conditions hold:

1. For any standard $g \in G$, we can obtain $h \in H$ such that $\iota(h, g)$ holds. Take a standard $g \in G$. By the surjectivity of lim, we have some $[f]$ such that $\lim[f] = g$. Transfer applies due to the standardness of lim and $g$, giving us $\mathrm{st}([f])$. Every standard equivalence class has a standard representative $f \in [f]$, so one can simply pick $h = f(\omega)$.

2. For any standard $g_1, g_2 \in G$, and $h \in H$ such that $\iota(h, g_1)$ and $\iota(h, g_2)$, we have $g_1 = g_2$. Take standard $g_1, g_2$ and arbitrary $h$ such that the approximations hold. Then we have $\lim[f_1] = g_1 \wedge f_1(\omega) = h$ and $\lim[f_2] = g_2 \wedge f_2(\omega) = h$ for respective $f_1, f_2$. Thanks to the equality $f_1(\omega) = h = f_2(\omega)$, and the fact that $\mathrm{st}(f_1), \mathrm{st}(f_2)$ both hold, we can apply Theorem 1.3.42 to conclude $[f_1] = [f_2]$ and therefore $g_1 = \lim[f_1] = \lim[f_2] = g_2$ as desired.

**Qed.**

**2.2.11.** Proposition 2.2.8 shows that Zilber's ordinary strong approximation (and the LEF condition, since the proof makes no essential use of lim) gives rise to a very special, well-behaved case of Definition 2.2.3. In particular, such an approximation has internal $\iota$ (but keep in mind that the predicate rarely has all parameters standard, so we generally cannot apply Transfer to it). The approximations of Proposition 2.2.7, which feature in the proofs of Theorems 1.2.7 and 1.3.45, do not share all the same good properties (as the reader should now verify). Nevertheless, these two constructions, along with Zilber's ordinary approximation (Proposition 2.2.10) all appear as special instances of our general definition. Moreover, we can see now that the language of Internal Set Theory enables us to consider finer-grained variations of these properties, some of which one cannot define easily (or investigate efficiently) in the ordinary setting. Definition 2.2.3 may seem overly general at first, but our main result (Theorem 2.3.9) does apply to every approximation. However, we also build a loose ranking of better-and-better-behaved approximations, and characterize some of the upper echelons of the resulting hierarchy.

**2.2.12.** We have yet to account for a significant detail: the problem of *structure preservation*. Proposition 2.2.8 does ensure that Zilber-style ordinary structural approximations give rise to approximations of the same set, but the result does not account for algebraic structure (in light of Proposition 2.2.7 a mere set-approximation result would hold little interest). In Proposition 2.2.13 we verify that the resulting approximation indeed preserves structure for the case of groups. The same holds for first-order theories in general, as long as one formulates the homomorphism property of the approximation predicate correctly ($\iota$ has to form a logical relation [21]) and the relevant quantifiers range over standard elements. We let the patient reader grapple with these details.

**2.2.13. Proposition.** Assume that we have a standard ordinary strong approximation of a group $G$ via the $I$-indexed sequence of groups $H$. Then $H_\omega$ approximates $G$ as a group for some index $\omega \in I$.

**Proof.** We only need to show that the predicate $\iota$ constructed in Proposition 2.2.8 satisfies $\forall h_1, h_2 \in H_\omega . \iota(h_1, g_1) \wedge \iota(h_2, g_2) \to \iota(h_1 h_2, g_1 g_2)$ for any standard $g_1, g_2 \in G$. We have a standard representative $(r \circ \mathrm{colim})(g_1) = f_1 \in \mathrm{colim}(g_1)$ such that $f_1(\omega) = h_1$. Similarly, we have a standard $f_2 \in \mathrm{colim}(g_2)$ such that $f_2(\omega) = h_2$, and a standard $f_3 \in \mathrm{colim}(g_1 g_2)$. We wish to prove $f_3(\omega) = h_1 h_2$. We know $\mathrm{st}(f_1 f_2)$ since the standardness of the multiplication operation follows from the standardness of the approximation. We also know that $f_1 f_2 \in \mathrm{colim}(g_1)\mathrm{colim}(g_2) = \mathrm{colim}(g_1 g_2)$ using the homomorphism property of colim. Consequently both $f_1 f_2$ and $f_3$ occur as standard representatives of the $\mathcal{D}$-equivalence class $\mathrm{colim}(g_1 g_2)$. It follows from Theorem 1.3.42 that two standard representatives have the same value at the monadic element $\omega$ of $\mathcal{D}$, and

therefore $f_3(\omega) = f_1(\omega)f_2(\omega) = h_1h_2$ as required.
**Qed.**

**2.2.14. Exercise.** Give another proof of Corollary 2.2.9 using the predicate $\iota(h,g) \leftrightarrow \exists^{st} f \in \mathrm{colim}(g).f(\omega) = h$. Which clauses of Proposition 2.2.8 does the resulting application satisfy?

**2.2.15.** We have to build up a library of approximations before we can "distill" the useful properties that approximations may have. We begin with well-known subclasses of the class of LEF groups. For Theorem 2.2.18 we have to briefly recall the definition of *profinite groups*. Similarly for Theorem 2.2.29 and *residually finite groups*.

## Profinite groups

**2.2.16. Definition.** The partially ordered set $(I, \leq)$ forms a *directed partial order* or *dpo* if every finite subset of $I$ has a $\leq$-upper bound in $I$.
Take a directed partial order $(I, \leq)$. An *inverse system of groups* over $(I, \leq)$ consists of an $I$-indexed set of groups $M$, and for every $i, j \in I$ with $i \leq j$ a group homomorphism $M_i^j : M_j \to M_i$.
Consider an inverse system $M$ over $(I, \leq)$. The set of functions

$$G = \left\{ f : (i \in I) \to M_i \,\middle|\, \forall i, j \in I. i \leq j \to M_i^j(f(j)) = f(i) \right\}.$$

forms a group when equipped with the pointwise group operations on $(i \in I) \to M_i$. We call this group the *inverse limit* of the system $M$ and denote it $\varprojlim M$.

**2.2.17. Definition.** If a group $G$ arises as an inverse limit of an inverse system of finite groups, we refer to $G$ as a *profinite group*.

**2.2.18. Theorem.** Every standard profinite group $G$ admits a finite approximation as a group.

**Proof.** Consider a standard profinite group $G$. We can find an inverse system of finite groups such that $G$ arises as an inverse limit of the system. Transfer applies, so we can in fact write $G$ as the inverse limit of a standard system of finite groups $M$ on a standard directed partial order $(I, \leq)$. For $i, j \in I$ with $i \leq j$, the system gives a group homomorphism $M_i^j : M_j \to M_i$. As per Definition 2.2.16 we can identify $G$ with a group of functions

$$G = \left\{ f : (i \in I) \to M_i \,\middle|\, \forall i, j \in I . i \le j \to M_i^j(f(j)) = f(i) \right\}$$

where we perform the group operation pointwise. Since $I$ forms a directed partial order, every standard finite subset of $I$ has an upper bound. Formally: $\forall^{st} F \subseteq I . \exists b . \forall i \in F . i \le b \wedge b \in I$. Idealization immediately yields $\exists \omega \in I . \forall^{st} i . i \le \omega$. Set $H = M_\omega$ and define the predicate $\iota(h, g)$ between $H$ and $G$ as an abbreviation for the formula $g(\omega) = h$. Then $\iota$ satisfies the following properties:

1. For any $g \in G$ we can find $h \in H$ such that $\iota(h, g)$ holds. Since $g : (i \in I) \to M_i$, we have $g(\omega) \in M_\omega = H$ and $\iota(g(\omega), g)$ holds trivially.

2. For any $g \in G$, $h_1, h_2 \in H$ such that $\iota(h_1, g)$ and $\iota(h_2, g)$ hold, we have $h_1 = h_2$. Just observe that $h_1 = g(\omega) = h_2$.

3. For any standard $g_1, g_2 \in G$ and any $h \in H$ such that $\iota(h, g_1)$ and $\iota(h, g_2)$ both hold, we have $g_1 = g_2$. Consider two standard functions $g_1, g_2 : (i \in I) \to M_i$. Assume that $g_1(\omega) = h = g_2(\omega)$. We prove that $g_1(i) = g_2(i)$ for all standard $i \in I$. Take any standard $i \in I$. We have $i \le \omega$ by construction of $\omega$, so the inverse system contains a group homomorphism $M_i^\omega : H \to M_i$ such that $M_i^\omega(g(\omega)) = g(i)$ holds for any $g \in G$. Applying $M_i^\omega$ to both sides of the equality $g_1(\omega) = g_2(\omega)$ yields $g_1(i) = g_2(i)$. Hence, $\forall^{st} i \in I . g_1(i) = g_2(i)$. By the standardness of $g_1, g_2$, Transfer applies to this statement and gives $g_1 = g_2$ as desired.

4. For any $g_1, g_2 \in G$ and $h_1, h_2 \in H$, if $\iota(h_1, g_1)$ and $\iota(h_2, g_2)$ both hold, then so does $\iota(h_1 h_2, g_1 g_2)$. We have $g_1(\omega) = h_1$ and $g_2(\omega) = h_2$. One can take products in $G$ pointwise, so $(g_1 g_2)(\omega) = g_1(\omega) g_2(\omega) = h_1 h_2$ as required.

**Qed.**

**2.2.19. Corollary.** Every profinite group admits an ordinary finite group approximation.

**Proof.** By the standardness of the conclusion, we can provisionally assume the standardness of the profinite group. By the definition of profinite groups, we can in fact write it as the inverse limit of a standard system of finite groups $M$ on a standard directed partial order $(I, \le)$. Choose $\omega, \iota$ as in Theorem 2.2.18, and construct a non-principal ultrafilter $\mathcal{D}$ with $\omega$ as its monadic element (follow the proof of Lemma 1.3.40).

Denote the ultraproduct $\prod_{i\in I} M_i/\mathcal{D}$ as $M_{\mathcal{D}}$ and consider the standard set

$$\lim = \left\{\!\!\left[ (G,f) \in M_{\mathcal{D}} \times \varprojlim M \;\middle|\; \forall^{st} g \in G.\exists h \in M_{\omega}.\iota(h,g) \wedge \forall^{st} i \in I.M_i^{\omega}(h) = f(i) \right]\!\!\right\}.$$

We claim that lim forms the graph of a surjective function $\lim : M_{\mathcal{D}} \to \varprojlim M$. First we prove that $\forall G \in M_{\mathcal{D}}.\exists f \in \varprojlim M.(G,f) \in \lim$. By Transfer it suffices to find standard $f \in \varprojlim M$ for standard $G \in M_{\mathcal{D}}$. A standard equivalence class $G$ has some standard representative $g \in G$. Pick such a representative and consider the set

$$f = \left\{(i,x) \in (i \in I) \times M_i \,\middle|\, M_i^{\omega}(g(\omega)) = x\right\}.$$

For any standard $i$ we have some $x$ such that $M_i^{\omega}(g(\omega)) = x$. By the standardness of the finite set $M_i$, Theorem 1.2.5 applies and gives $\mathrm{st}(x)$. Using Transfer, this shows that $f : (i \in I) \to M_i$. Similarly, for all standard $i \le j \in I$ we have $M_i^j(f(j)) = M_i^j(M_j^{\omega}(g(\omega))) = M_i^{\omega}(g(\omega)) = f(i)$. Using Transfer one more time, we conclude $f \in \varprojlim M$. To demonstrate that $\forall^{st} g \in G.\forall^{st} i \in I.M_i^{\omega}(g(\omega)) = f(i)$, consider any other standard $g'$ and construct a corresponding function $f'$. Since $\mathrm{st}(g)$ and $\mathrm{st}(g')$ both hold, and $[g] = [g']$, Theorem 1.3.42 immediately gives $g(\omega) = g'(\omega)$, and we conclude that $f$ and $f'$ have the same standard values. But Transfer applies, so $f = f'$.
Consider a standard $f \in \varprojlim M$. We have $f : (i \in I) \to M_i$ and hence $[f] \in M_{\mathcal{D}}$. Since $f \in [f]$, we have $\forall^{st} g \in [f].g(\omega) = f(\omega)$, and therefore we know that $\forall^{st} f \in \varprojlim M.\forall^{st} i \in I.\lim([f])(i) = f(i)$. The usual Transfer argument goes through, and we get $\lim([f]) = f$ for all $f \in \varprojlim M$, proving the surjectivity of lim.
Finally, we must show that lim respects the group operation. As before, having made the usual provisional assumptions, we only need to show that for standard $G, H \in M_{\mathcal{D}}$, $\lim(G)\lim(H)$ and $\lim(GH)$ take the same value on all standard $i \in I$.
Consider any standard $g \in G$ and $h \in H$, and some standard $i \in I$ We have $st(GH), \mathrm{st}(gh)$, $gh \in GH$ and

$$\begin{aligned}
\lim(GH)(i) &= M_i^{\omega}(gh(\omega)) \\
&= M_i^{\omega}(g(\omega)h(\omega)) \\
&= M_i^{\omega}(g(\omega))M_i^{\omega}(h(\omega)) \\
&= \lim(G)(i)\lim(H)(i).
\end{aligned}$$

We have shown that for any standard profinite group $\varprojlim M$ with $I, \le, M$ standard we have a a non-principal ultrafilter $\mathcal{D}$ and a surjective group homomorphism $M_{\mathcal{D}} \to$

$\varprojlim M$. By the internality of this conclusion, we can drop the provisional assumptions of standardness and conclude that any standard profinite group admits an ordinary group approximation.
**Qed.**

**2.2.20. Proposition.** Every standard profinite group $(G, \multimapinv)$ admits a finite approximation as a group in the profinite topology $\multimapinv$.

**Proof.** Exercise. Hint: The approximation of Theorem 2.2.18 does the job. To prove this, observe that $f_1 \multimapinv f_2$ in the profinite topology implies having the same values at standard arguments.
**Qed.**

## Properties of approximations

**2.2.21.** We have seen in Proposition 2.2.13 that the approximations constructed from ordinary approximations using Proposition 2.2.8 preserve the group operation. The approximations coming from Theorem 2.2.18 possess even better properties than the ones obtained from ordinary approximations in Proposition 2.2.13 (not to mention the approximations of Exercise 2.2.14). Not only does our approximation have an internal $\iota$ that acts like a group homomorphism on standard elements, but the homomorphism property obtains for *any* pair of elements, even non-standard ones. At this stage, talking about all these different properties starts to feel tedious: time to consolidate what we learned into a few memorable adjectives. We state Definition 2.2.22 for the case of groups only: algebraic structures work the same way, and the interested reader can contend with the general first-order case as in 2.2.12.

**2.2.22. Definition.** Consider two groups $H, G$ where $H$ approximates $G$ as a group via the predicate $\iota$. We call $\iota$

1. *internal* if $\iota$ forms an internal predicate;
2. *entire* if for any $g \in G$ we can find $h \in H$ such that $\iota(h, g)$ holds;
3. *robust* if for any $g_1, g_2 \in G$ and $h_1, h_2 \in H$ such that $\iota(h_1, g_1)$ and $\iota(h_2, g_2)$ both hold, we have $\iota(h_1 h_2, g_1 g_2)$.

**2.2.23.** The approximations constructed in the proof of Proposition 2.2.13 are internal and entire but usually not robust. The approximations constructed as part of Proposition 2.2.10 and Exercise 2.2.14 are neither internal nor entire, but they are both robust. The approximations coming from Theorem 2.2.18 are internal, entire and robust. Using specific properties of the first-order theory under consideration allows us to relate ordinary strong approximations of certain structures with robust approximations of the same structures: Theorem 2.2.24 gives an example.

**2.2.24. Theorem.** Take any field $\mathbb{F}$. Assume that we have a standard ordinary approximation of an $\mathbb{F}$-vector-space $G$ via the $I$-indexed sequence of $\mathbb{F}$-vector-spaces $H$. Then we can find an internal, entire, robust approximation of $G$ via $H_\omega$ for some index $\omega \in I$.

**Proof.** Denote the $I$-ultrafilter coming from the ordinary approximation as $\mathcal{D}$. Taking a direct product of vector spaces yields another vector space, so we can regard the quotient map $[-] : \prod_{i\in I} H_i \to \prod_{i\in I} H_i/\mathcal{D}$ as an epimorphism in the category of $\mathbb{F}$-vector-spaces. But every epimorphism splits in that category, so we get a linear transformation $r : \prod_{i\in I} H_i/\mathcal{D} \to \prod_{i\in I} H_i$ such that $\forall x.([-]\circ r)(x) = x$. By Proposition 1.3.38 the ultrafilter $\mathcal{D}$ has a monadic element $\omega \in I$. Define $\imath(h, g)$ between $H_\omega$ and $G$ as an abbreviation for the formula $(r\circ \mathrm{colim})(g)(\omega) = h$. The resulting approximation $\imath$ has the internal and entire properties by Proposition 2.2.8. We only need to prove robustness.
Consider any $g_1, g_2 \in G$ and $h_1, h_2 \in H_\omega$ such that $\imath(h_1, g_1)$ and $\imath(h_2, g_2)$ both hold. Take any scalar $\lambda \in \mathbb{F}$. We have $(r\circ \mathrm{colim})(g_1)(\omega) = h_1$ and $(r\circ \mathrm{colim})(g_2)(\omega) = h_2$. We need to prove that $(r\circ \mathrm{colim})(\lambda g_1 + g_2)(\omega) = \lambda h_1 + h_2$. Using the linearity of both colim and $r$, we have $(r\circ \mathrm{colim})(\lambda g_1 + g_2) = r(\lambda \mathrm{colim}(g_1) + \mathrm{colim}(g_2)) = \lambda r(\mathrm{colim}(g_1)) + r(\mathrm{colim}(g_2))$ as members of the function space $\prod_{i\in I} H_i$. Equality of functions implies pointwise equality on all indices, so taking the index $\omega \in I$ gives us $(r\circ \mathrm{colim})(\lambda g_1 + g_2)(\omega) = \lambda(r\circ \mathrm{colim}(g_1))(\omega) + (r\circ \mathrm{colim}(g_2))(\omega) = \lambda h_1 + h_2$, which shows the robustness of the approximation.
**Qed.**

**2.2.25.** One cannot imitate the reasoning of Theorem 2.2.24 in the case of arbitrary groups. Given an ordinary strong approximation of $G$ via the sequence of finite groups $H_i$, one wishes to find a section $r : \prod_{i\in I} H_i/\mathcal{D} \to \prod_{i\in I} H_i$ for the quotient map $[-] : \prod_{i\in I} H_i \to \prod_{i\in I} H_i/\mathcal{D}$. Upon success, we would have $r\circ \mathrm{colim} : G \hookrightarrow \prod_{i\in I} H_i$. Only residually finite groups $G$ admit such a morphism. This does not mean that robust approximation implies residually finiteness, merely that we cannot use the construction of Proposition 2.2.13 to find robust approximations in the non-residually-finite case.

## Residually finite groups

**2.2.26. Definition.** We call a group $G$ *residually finite* if it embeds into some direct product of finite groups.

**2.2.27. Proposition.** A group $G$ is residually finite precisely if for any finite subset $F \subseteq G$ with $1 \notin F$ we can find a finite index normal subgroup $N$ of $G$ such that $\forall x \in F . x \notin N$.

**Proof.** See [7]-Corollary 2.2.6.
**Qed.**

**2.2.28. Corollary.** A standard group $G$ satisfies residually finiteness precisely if it contains a finite index normal subgroup $N$ that does not have any standard element (apart from the identity).

**Proof.** Apply Idealization to the normal subgroup condition of Proposition 2.2.27.
**Qed.**

**2.2.29. Theorem.** Every standard residually finite group $G$ admits a finite internal, entire, robust approximation.

**Proof.** Take a residually finite group $G$. We can use Corollary 2.2.28 to choose a finite index subgroup $N$ that does not contain any standard (non-identity) element of $G$. Taking the quotient $H = G/N$ yields a finite group by the finite index clause. Let $\iota(h, g)$ stand for the binary predicate $g \in h$ for $g \in G$ and $h \in G/N$. Internality follows by the form of $\iota$. We prove the other clauses below:

1. For any $g \in G$, we have $h \in H$ so $\iota(h, g)$ holds. We can take $h = gN$, thereby showing that $\iota$ is entire.

2. For any $g \in G$ and $h_1, h_2 \in H$ such that $g \in h_1$ and $g \in h_2$, we have $h_1 = h_2$. This just restates the fact that left cosets of a normal subgroup are either disjoint or identical.

3. For any standard $g_1, g_2 \in G$, and $h \in H$ such that $g_1 \in h$ and $g_2 \in h$, we have $g_1 = g_2$. Writing $h = xN$, we get $g_1 x^{-1} \in N$ and $x^{-1} g_2^{-1} \in N$, and thus $g_1 g_2^{-1} \in N$. But $\mathrm{st}(g_1 g_2^{-1})$ holds by Corollary 1.2.11, and the only standard element of $N$ equals the identity. Hence $g_1 g_2^{-1} = 1$ and so $g_1 = g_2$.

4. For any $g_1, g_2 \in G$ and any $h_1 \in H, h_2 \in H$ such that $g_1 \in h_1$ and $g_2 \in h_2$ we have $g_1 g_2 \in g_1 g_2 N = h_1 h_2$ by the definition of multiplication in $G/N$.

**Qed.**

**2.2.30.** Theorem 2.2.29 proves Theorem 2.2.18 as a special case, since every profinite group has the property of residually finiteness. However, one does not get a simple proof of the topological approximation result (Proposition 2.2.20) this way.

**2.2.31.** Unlike the construction of Theorem 2.2.18, which gave as a corollary the ordinary approximability of profinite groups (Corollary 2.2.19), we cannot expect the result of Theorem 2.2.29 to transfer to ordinary approximations: we construct a counterexample in Proposition 2.2.32.

**2.2.32. Proposition.** Some residually finite groups do not admit ordinary finite approximations.

**Proof.** Assume for a contradiction that all residually finite groups admit ordinary finite approximations. Regard $SO(3)$ as a quotient $f : F \twoheadrightarrow SO(3)$ of the free group generated by all matrices in $SO(3)$. Since free groups are residually finite, our assumption allows us to find a finite ordinary approximation $\lim : \prod_{i \in I} H_i/\mathcal{D} \twoheadrightarrow F$, and to get an approximation of $SO(3)$ as $f \circ \lim : \prod_{i \in I} H_i/\mathcal{D} \twoheadrightarrow SO(3)$. This contradicts [34]-Theorem 7 on ordinary approximations of $SO(3)$.
**Qed.**

## The Alexandroff case

**2.2.33.** We wish to investigate the "best possible" approximations: ones where the approximation predicate $\iota$ constitutes a genuine, bona fide homomorphism of groups. Analogizing with Alexandroff spaces, which arise as the spaces where the nearness predicate forms a bona fide relation, we call these *Alexandroff approximations*. Here we classify groups that admit such finite approximations. One can define Alexandroff approximation for other algebraic structures analogously; the diligent reader would fill in these details while attempting Exercise 2.2.37.

**2.2.34. Definition.** Consider an approximation $\iota$ of a group $G$ (equipped with some Fréchet predicate $\multimapdotinv$) via a finite group $H$. We call $\iota$ an *Alexandroff approximation* if $\iota(h, g) \leftrightarrow f(h) = g$ for some group homomorphism $f : H \to G$.

**2.2.35. Definition.** We call a group $G$ *locally finite* if every finitely generated subgroup of $G$ has finite order.

**2.2.36. Proposition.** A group $(G, \multimap)$ admits a finite Alexandroff approximation precisely if $G$ is locally finite. In that case one can find an Alexandroff approximation by a subgroup $H < G$.

**Proof.** Provisionally assume that $G$ is standard. Assume that $G$ admits some Alexandroff approximation $\imath$ via $H$. Consider the group homomorphism $f : H \to G$, and take the image $f(H) < G$. This subgroup clearly has finite order. Since $\forall^{st} g \in G.\exists h \in H.\imath(h,g)$ holds, we have that $\forall^{st} g \in G.g \in f(H)$, and therefore we can find some $X < G$ that forms a finite subgroup of $G$ that contains every standard element of $G$. Using Proposition 1.2.4 we get that for any standard finite subset $F \subseteq G$ we can find a finite subgroup $X < G$ such that $F \subseteq X$. Thus $\langle F \rangle \subseteq X$, and so $\langle F \rangle$ has finite order. By Transfer the same holds for every finite subset of $G$.
Now assume that every finitely generated subgroup of $G$ has finite order. Take a finite subset $H$ of $G$ that contains every standard element of $G$ (follow e.g. the proof of Proposition 2.2.7). Since $\langle H \rangle$ has finite order, it constitutes a finite subgroup that nonetheless contains every standard element of $G$. Take the inclusion map $f : \langle H \rangle \hookrightarrow G$ and set $\imath(h,g)$ as an abbreviation for $f(h) = g$. We have to verify three properties:

1. For standard $g \in G$ we can find $h \in \langle H \rangle$ with $f(h) = g$. Since $g \in H \subseteq \langle H \rangle$, we can set $h = g$ and have $f(h) = h = g$.

2. For standard $g \in G$, arbitrary $h_1, h_2 \in \langle H \rangle$ such that $f(h_1) = g$ and $f(h_2) = g$, we have $h_1 = h_2$. Since we obtained $f$ as an inclusion map, we have $h_1 = g = h_2$ as desired.

3. For any $g_1, g_2 \in G$ and $h \in \langle H \rangle$ such that $f(h) = g_1$ and $f(h) = g_2$ we have $g_1 \multimap g_2$. In this case we have $g_1 = g_2$, so by reflexivity $g_1 \multimap g_2$.

4. For $g_1, g_2 \in G$ and $h_1, h_2 \in \langle H \rangle$ such that $f(h_1) = g_1$ and $f(h_2) = g_2$, we have $f(h_1 h_2) = g_1 g_2$. Again, this holds simply because we have $h_1 h_2 \in \langle H \rangle$, $h_1 = g_1$ and $h_2 = g_2$.

**Qed.**

**2.2.37. Exercise.** Prove that every Boolean algebra admits an Alexandroff approximation.

**2.2.38.** The correspondence presented in Proposition 2.2.36 sheds light on the "unreasonable effectiveness" of Internal Set Theory for locally finite structures. Even Theorem 1.2.7 relies essentially on the locally finiteness of graph structures: taking the subgraph induced by a finite subset of vertices results in a finite subgraph. As a more open-ended exercise, applying Exercise 2.2.37 to Goldblatt's superstructure proof of the Stone representation theorem ([16]-Chapter 19.6.) yields a very legible IST-proof of the same fact.

## 2.3 Action extension

**2.3.1.** Approximations allows us to extend well-behaved functions defined on the approximating object to similarly well-behaved functions defined on the approximated object, as long as the codomain of the function comes equipped with a nice topology. In particular, we show that if a group admits a finite approximation with a Lipschitz action on some compact manifold, then we can lift this action and obtain an action of any periodic subgroup of the approximated group on the same manifold (Theorem 2.3.9).

**2.3.2. Proposition.** Let $\imath$ be a weak approximation of the standard set $G$ via the (not necessarily finite) set $H$. Consider a standard compact Hausdorff topological space $(M, \multimapinv)$ and a function $f : H \to M$ such that $\forall h_1, h_2 \in H. \forall g \in G. \imath(h_1, g) \wedge \imath(h_2, g) \to f(h_1) \multimapinv f(h_2)$. There is a function $f' : G \to M$ such that for any standard $g \in G$, if $\imath(h, g)$ then $f'(g) \multimapinv f(h)$.

**Proof.** Take such a function $f : H \to M$. Define the set $f'$ via the Standardization axiom as $f' = \{\!\![ (g, m) \in G \times M \mid \exists h \in H. \imath(h, g) \wedge m \multimapinv f(h) ]\!\!\}$. We claim that $f'$ forms the graph of a function $f' : G \to M$.

We first prove that $\forall^{st} g. \exists^{st}! m \in M.(g, m) \in f'$. For existence, take a standard $g \in G$. Since $\imath$ weakly approximates $G$ via $H$, we can find $h \in H$ such that $\imath(h, g)$. Using the compactness of $M$, we immediately get a standard $m \in M$ such that $m \multimapinv f(h)$. Thus we have $(g, m) \in f'$. For uniqueness, take standard $g \in G, m_1 \in M$ and $m_2 \in M$, and assume $(g, m_1) \in f'$ and $(g, m_2) \in f'$. By definition we get $h_1, h_2 \in H$ such that $\imath(h_1, g), m_1 \multimapinv f(h_1)$ and $\imath(h_2, g), m_2 \multimapinv f(h_2)$ all hold. From $\imath(h_1, g)$ and $\imath(h_2, g)$ we obtain $f(h_2) \multimapinv f(h_1)$, and hence (by the properties of the universal representation $\multimapinv$) $m_1 \multimapinv f(h_2)$. Now we have $m_1 \multimapinv f(h_2)$ and $m_2 \multimapinv f(h_2)$, so by the Hausdorff property we conclude $m_1 = m_2$.

Notice that $\mathrm{st}(f')$ holds by Standardization. This means that Transfer applies to the

formula $\forall^{st} g.\exists^{st}!m \in M.(g,m) \in f'$, which proves our claim that $f' : G \to M$. For any standard $g \in G$ we have $\mathrm{st}(f'(g))$, and since $(g, f'(g)) \in f'$ we get $\forall h \in H.\imath(h,g) \to f'(g) \multimapinv f(h)$ as desired.
**Qed.**

**2.3.3. Corollary.** Consider an approximation $\imath$ of the standard set $G$ via the (not necessarily finite) set $H$. For any function $f : H \to M$ to a standard compact Hausdorff space $M$ we can find a standard function $f' : G \to M$ such that for all standard $g \in G$, if $\imath(h,g)$ then $f'(g) \multimapinv f(h)$.

**Proof.** By the definition of approximation we have $\imath(h_1,g) \wedge \imath(h_2,g) \to h_1 = h_2$ for any standard $g \in G$. Consequently, every function $f : H \to M$ satisfies $f(h_1) = f(h_2)$, and a fortiori $f(h_1) \multimapinv f(h_2)$. We get the claimed result by applying Proposition 2.3.2.
**Qed.**

**2.3.4.** Notice that one cannot weaken the compactness requirement in either Proposition 2.3.2 or Corollary 2.3.3. In fact, given a non-compact $M$, we can use the characterization of Theorem 1.3.33 to find a point $m$ that lies far from every standard point, i.e. $\forall^{st} x \in M.\neg x \multimapinv m$. Then we cannot even extend the constant function $f(x) = m$. One can see the failure of the extension results for non-compact $M$ as a (very loose) counterpart to [51]-Proposition 3.4.

**2.3.5.** We are now ready to prove our main result for this chapter, Theorem 2.3.9, which relates actions of an approximating group $H$ on standard manifolds to actions of periodic subgroups of the approximated group $G$ on the same manifold. One can see Theorem 2.3.9 as a (vast) generalization of a result on discrete circle actions due to Manevitz and Weinberger [30]. The group-theoretic underpinning of our result comes from the celebrated Newman's theorem on group actions, which states that a compact Lie group does not act on a manifold with uniformly small orbits. For what follows recall that we label a group *periodic* if each element of the group has finite order.

**2.3.6. Theorem** (Newman)**.** Take a manifold $M$ metrized by the metric $d$, and consider a non-empty open subset $U \subseteq M$. We can find a real number $\nu > 0$ depending only on $U$ and the restriction of $d$ to $U$ such that for any compact Lie group $G$, the only continuous action $\circlearrowleft : G \times M \to M$ satisfying $d(x, g \circlearrowleft x) \leq \nu$ for all $g \in G$ and $x \in U$ is the trivial action.

**Proof.** See [37]-Theorem 1.

**Qed.**

**2.3.7. Corollary.** For any standard compact metric manifold $M$ equipped with a standard metric, we have a standard real number $\nu > 0$ such that for any finite group $G \neq 1$, continuous faithful action $\circlearrowright : G \times M \to M$ and element $g \in G$, we can find $n \in \mathbb{N}$ and $x \in M$ with $d(g^n \circlearrowright x, x) > \nu$.

**Proof.** Consider a standard compact metric manifold $M$. Set $U = M$ and obtain a $\nu > 0$ from Theorem 2.3.6. We can pick a standard such $\nu$ by Transfer. Take the finite group $\langle g \rangle < G$. By faithfulness the restricted action $\circlearrowright|_{\langle g \rangle} : \langle g \rangle \times M \to M$ is non-trivial, and so by Theorem 2.3.6 there are $h \in \langle g \rangle$ and $x \in M$ such that $d(h \circlearrowright x, x) > \nu$. Since $h$ belongs to the finite group $\langle g \rangle$ we can write $h = g^n$ for some $n \in \mathbb{N}$.
**Qed.**

**2.3.8. Definition.** Take a positive constant $K \in \mathbb{R}$. Consider a group $G$, a metric space $(M, d)$ and an action $\circlearrowright : G \times M \to M$. We call this action $K$-Lipschitz if for all $g \in G$, $x, y \in M$, we have $d(g \circlearrowright x, g \circlearrowright y) \leq K \cdot d(x, y)$.

**2.3.9. Theorem** (Main Result)**.** Consider a standard group $G$ approximated by the finite group $H$, and a standard compact manifold $M$. Assume that the group $H$ acts faithfully on the manifold $M$ via the action $\circlearrowright : H \times M \to M$, and that this action is $K$-Lipschitz for some standard $K > 0$ (on some metrization of the manifold). Then every periodic subgroup of $G$ admits a faithful $K$-Lipschitz action on $M$.

**Proof.** For the sake of readability, we divide this long proof into multiple claims.

**Claim 1**: The set $G \times M$ approximates $H \times M$.

Define the predicate $\iota'((h, m_h), (g, m_g))$ between $H \times M$ and $G \times M$ as an abbreviation for $\iota(h, g) \wedge m_h = m_g$. We need to prove the usual properties.

1. For any standard $(g, m) \in G \times M$ we can find $(h, m_h) \in H \times M$ such that we have $\iota'((h, m_h), (g, m))$. Take standard $(g, m) \in G \times M$. Since $\iota$ satisfies the analogous property, choose $h \in H$ such that $\iota(h, g)$. We clearly have $\iota((h, m), (g, m))$.

2. For any standard $(g, m) \in G \times M$ and arbitrary $(h_1, m_1), (h_2, m_2) \in H \times M$ such that $\iota'((h_1, m_1), (g, m))$ and $\iota'((h_2, m_2), (g, m))$ both hold, we have $(h_1, m_1) = (h_2, m_2)$. The assumptions guarantee $m_1 = m = m_2$, so we only need to prove $h_1 = h_2$. This follows since both $\iota(h_1, g)$ and $\iota(h_2, g)$ hold and $\iota$ satisfies the analogous property.

3. For any standard $(g_1, m_1), (g_2, m_2) \in G \times M$ and arbitrary $(h, m) \in H \times M$ such that $\iota'((h,m),(g_1,m_1))$ and $\iota'((h,m),(g_2,m_2))$ both hold, we have $(g_1,m_1) = (g_2,m_2)$. Again, the assumptions guarantee $m_1 = m = m_2$, so we only need to prove $g_1 = g_2$. This follows from $\iota(h, g_1)$ and $\iota(h, g_2)$ using the analogous property of $\iota$.

In what follows, fix a standard metrization of the manifold $M$, and hence realize it as a compact metric space $(M, d)$. Denote the nearness predicate coming from Proposition 1.3.35 as $\multimapinv$.

> **Claim 2**: We can find a standard function $\circlearrowright' : G \times M \to M$ such that for all standard $g \in G$ and $m \in M$, if we have $\iota(h, g)$ then we also have $g \circlearrowright' m \multimapinv h \circlearrowright m$.

Using Claim 1, we know that $\iota'$ approximates $G \times M$ via $H \times M$. Moreover, $M$ forms a compact Hausdorff topological space with the nearness predicate $\multimapinv$. Applying Corollary 2.3.3 to the function $\circlearrowright : G \times M \to M$ gives us a function $\circlearrowright' : G \times M$ such that for any standard $(g, m) \in G \times M$ and any $(h, m_h) \in H \times M$ with$\iota'((h, m_H), (g, m))$, we have $g \circlearrowright' m \multimapinv h \circlearrowright m_H$. By the definition of $\iota'$, we have $m_H = m$. Thus, if we have $\iota(h, g)$ then we also have $g \circlearrowright' m \multimapinv h \circlearrowright m$.

> **Claim 3**: We have $\iota(1_H, 1_G)$.

Recall that Definition 2.2.3 mandates only the preservation of the group operation; we prove the preservation of the identity element as a consequence. Since $\mathrm{st}(1_G)$ holds, we have some $h \in H$ such that $\iota(h, 1_G)$. It suffices to prove that $h = 1_H$. From $\iota(h, 1_G)$ we get $\iota(h^2, 1_G^2)$ using the fact that $\iota$ preserves the group operation. But then $\iota(h^2, 1_G)$, and from $\mathrm{st}(1_G)$ we get $h^2 = h$. Multiplying both sides by $h^{-1}$ yields $h = 1_H$ as desired.

> **Claim 4**: The action $\circlearrowright : H \times M \to M$ satisfies uniform S-continuity, i.e. for all $h \in H$ and $x, y \in M$, if $x \multimapinv y$ then $h \circlearrowright x \multimapinv h \circlearrowright y$.

We start by proving that this holds for standard $x \in M$. So consider arbitrary $h \in H$, standard $x \in M$ and arbitrary $y \in M$ with $x \multimapinv y$. Take any standard $\varepsilon > 0$. Since $\mathrm{st}(K)$ holds, we know that $\mathrm{st}(K^{-1}\varepsilon)$ holds as well. By $x \multimapinv y$, we have $d(x, y) < s$ for any standard $s > 0$. In particular $d(x, y) \le K^{-1}\varepsilon$. By the $K$-Lipschitz property of the action $\circlearrowright$, we know that $d(h \circlearrowright x, h \circlearrowright y) < K d(x, y) \le K K^{-1} \varepsilon = \varepsilon$. Since we chose an arbitrary standard $\varepsilon > 0$, we get $h \circlearrowright x \multimapinv h \circlearrowright y$ as desired.
Now we must prove the same for arbitrary $x \in M$. Consider arbitrary $h \in H$, $x \in M$ and $y \in M$ with $x \multimapinv y$. Use the compactness of $M$ to pick standard $x'$ such that $x' \multimapinv x$. By transitivity we have $x' \multimapinv y$ as well, so by the previous result we have both $h \circlearrowright x' \multimapinv h \circlearrowright x$ and $h \circlearrowright x' \multimapinv h \circlearrowright y$. From symmetry and transitivity it follows that $h \circlearrowright x \multimapinv h \circlearrowright y$.

**Claim 5**: For each $m \in M$ we have $1_G \circlearrowleft' m = m$.

We know that $\mathrm{st}(\circlearrowleft')$, so we can provisionally assume the standardness of $m$. In that case we have $1_G \circlearrowleft' m \multimapinv h \circlearrowleft m$ for each $h \in H$ with $\iota(h, 1_G)$ by Claim 2. From Claim 3 we know that $\iota(1_H, 1_G)$, so $1_G \circlearrowleft' m \multimapinv 1_H \circlearrowleft m = m$. Using $\mathrm{st}(m)$ it follows that $1_G \circlearrowleft' m = m$ by the Hausdorff property of $M$.

**Claim 6**: For each $g, h \in G$ and $m \in M$ we have $gh \circlearrowleft' m = g \circlearrowleft' (h \circlearrowleft' m)$.

Caveat: in this part $h$ belongs to $G$, not to $H$! Given the internal conclusion, we provisionally assume the standardness of $g, h$ and $m$. Since $\mathrm{st}(g), \mathrm{st}(h)$ hold we can find $g' \in H$ and $h' \in H$ such that $\iota(g', g)$ and $\iota(h', h)$. By preservation of the group operation for standard elements, we have $\iota(g'h', gh)$ as well. On one hand, we have

$$gh \circlearrowleft' m \multimapinv g'h' \circlearrowleft m = g' \circlearrowleft (h' \circlearrowleft m).$$

On the other hand, we have $g \circlearrowleft' (h \circlearrowleft' m) \multimapinv g' \circlearrowleft (h \circlearrowleft' m)$. Why? Because $h$ and $m$ are standard, and hence $\mathrm{st}(h \circlearrowleft' m)$, so Claim 2 applies. We also have $h \circlearrowleft' m \multimapinv h' \circlearrowleft m$. Applying Claim 4 immediately yields $g' \circlearrowleft (h \circlearrowleft' m) \multimapinv g' \circlearrowleft (h' \circlearrowleft m)$, so we get

$$g \circlearrowleft' (h \circlearrowleft' m) \multimapinv g' \circlearrowleft (h' \circlearrowleft m).$$

Notice that both $gh \circlearrowleft' m$ and $g \circlearrowleft' (h \circlearrowleft' m)$ satisfy standardness. We have shown that these two standard elements have a common neighbor. Therefore, by Hausdorff separation (Definition 1.3.25) we conclude $gh \circlearrowleft' m = g \circlearrowleft' (h \circlearrowleft' m)$, which proves our claim, and with Claim 5 proves that $\circlearrowleft : G \times M \to M$ forms an action of $G$ on $M$.

**Claim 7**: The action $\circlearrowleft' : G \times M \to M$ has Lipschitz constant $K$.

By the internality of the conclusion, we can provisionally assume the standardness of everything in sight. So take standard $g \in G$, $x, y \in M$. We wish to prove $d(g \circlearrowleft' x, g \circlearrowleft' y) \leq K d(x, y)$. Pick $g'$ with $\iota(g', g)$ and any standard $\varepsilon > 0$. Observe that by Claim 2, we have $d(g \circlearrowleft' x, g' \circlearrowleft x) < \frac{\varepsilon}{2}$ and similarly for $y$. By repeated applications of the triangle inequality, we get that

$$\begin{aligned}
d(g\circlearrowright' x, g\circlearrowright' y) &\leq d(g\circlearrowright' x, g'\circlearrowright y) + d(g'\circlearrowright y, g\circlearrowright' y) \\
&\leq d(g\circlearrowright' x, g'\circlearrowright y) + \frac{\varepsilon}{2} \\
&\leq d(g'\circlearrowright x, g'\circlearrowright y) + d(g'\circlearrowright x, g\circlearrowright' x) + \frac{\varepsilon}{2} \\
&\leq d(g'\circlearrowright x, g'\circlearrowright y) + \frac{\varepsilon}{2} + \frac{\varepsilon}{2} \\
&= d(g'\circlearrowright x, g'\circlearrowright y) + \varepsilon \\
&\leq K d(x, y) + \varepsilon.
\end{aligned}$$

and therefore $d(g\circlearrowright' x, g\circlearrowright' y) - Kd(x, y) \leq \varepsilon$ for all standard $\varepsilon > 0$. By Transfer we immediately obtain $d(g\circlearrowright' x, g\circlearrowright' y) \leq Kd(x, y)$. Discharging the provisional standardness assumptions, we conclude that the action $\circlearrowright'$ admits the standard Lipschitz constant $K$ as we claimed.

> **Claim 8**: The action $\circlearrowright : H \times M \to M$ satisfies (metric, $\varepsilon$-$\delta$) continuity.

Note that Claim 8 does not follow from Claim 4, as we do not have $\mathrm{st}(\circlearrowright)$ (the reader has already constructed counterexamples as part of Exercise 1.3.20). For each $h \in H$, $x \in M$ and $\varepsilon > 0$ we need to find some $\delta > 0$ such that for $y \in M$, $d(x, y) < \delta$ implies $d(h\circlearrowright x, h\circlearrowright y) < \varepsilon$. But this follows immediately from the existence of the Lipschitz constant, by taking $\delta = K^{-1}\varepsilon$.

> **Claim 9**: Consider any periodic subgroup $X < G$ and $g \in X$ such that $g \neq 1$. We have $g\circlearrowright' m \neq m$.

By the internality of the conclusion, we can provisionally assume the standardness of both the subgroup $X$ and the element $g \in X$. Given the periodicity of $X$, the element $g$ has finite order. Moreover, $\mathrm{st}(x)$ holds, and therefore Proposition 1.2.9 guarantees the standardness of $\mathrm{ord}(x) \in \mathbb{N}$.
Consider $h \in H$ for which we have $\iota(h, g)$. Then for any standard $k \in \mathbb{N}$, we have $\iota(h^k, g^K) \to \iota(h^{k+1}, g^{k+1})$. Thus, by the principle of External Induction (Theorem 1.2.15) we get that $\iota(h^n, g^n)$ for all standard $n \in \mathbb{N}$. In particular, for $n = \mathrm{ord}(x)$ we have $\iota(h^n, 1_G)$. We already know $\iota(1_H, 1_G)$ from Claim 3, so we can conclude $h^n = 1_H$. Consequently, $\mathrm{ord}(h) \leq \mathrm{ord}(g)$ and by Proposition 1.2.14 we obtain that $h$ has standard order.
We now apply Corollary 2.3.7 of Newman's theorem to the group $H$, the element $h$ and the action $\circlearrowright : H \times M \to M$. For this we need to use Claim 8. We get a standard $\nu > 0$,

some $n \in \mathbb{N}$ and some $m' \in M$ such that $d(h^n \circlearrowright m', m') > \nu$, and therefore $\neg(h^n \circlearrowright m' \multimapinv m')$.
We know that $\mathrm{st}(n)$ holds, since $n < \mathrm{ord}(h)$ and we have proved the standardness of $\mathrm{ord}(h)$ above. Unfortunately, we cannot expect $m' \in M$ to satisfy standardness. However, using the compactness of $M$ we can obtain a standard $m \in M$ with $m \multimapinv m'$. We prove that $\neg(h^n \circlearrowright m \multimapinv m)$. Assume for a contradiction that $h^n \circlearrowright m \multimapinv m$ does hold. As we have $h^n \circlearrowright m \multimapinv h^n \circlearrowright m'$ from Claim 4, we could use the symmetry and transitivity of the predicate $\multimapinv$ to get $h^n \circlearrowright m' \multimapinv h^n \circlearrowright m \multimapinv m \multimapinv m'$ and reach a contradiction.
By the previous External Induction argument, we know $\imath(h^n, g^n)$, and Claim 2 gives us $g^n \circlearrowright' m \multimapinv h^n \circlearrowright m$. Having $g^n \circlearrowright' m \multimapinv m$ would lead to the contradictory chain $h^n \circlearrowright m \multimapinv g^n \circlearrowright' m \multimapinv m$, so $\neg(g^n \circlearrowright' m \multimapinv m)$. We conclude $g^n \circlearrowright' m \neq m$ using the reflexivity of the nearness predicate $\multimapinv$. Since $g^n \circlearrowright' m \neq m$, clearly $g \circlearrowright' m \neq m$.
Transfer allows us to dispense with the provisional assumptions of standardness on $X$ and $g$, so for every element $g$ of any periodic subgroup $X$ we can find $m \in M$ with $g \circlearrowright' m \neq m$. We conclude that each periodic subgroup $X$ of the approximated group $G$ acts faithfully on the manifold $M$ via the map $\circlearrowright' : X \times M \to M$. So concludes the proof of our main result.
**Qed.**

**2.3.10. Corollary.** Consider a standard group $G$ approximated by the finite group $H$, and a standard compact manifold $M$. Assume that the group $H$ acts isometrically on the manifold $M$ via some action $\circlearrowright : H \times M \to M$. Then every periodic subgroup $X < G$ admits an isometric action on $M$.

**Proof.** An isometric action satisfies the $K$-Lipschitz condition for $K = 1$. We can obtain the action $\circlearrowright' : G \times M \to M$ as we do in Theorem 2.3.9. We already know that $d(g \circlearrowright' x, g \circlearrowright' y) \leq d(x, y)$, so proving $d(x, y) \leq d(g \circlearrowright' x, g \circlearrowright' y)$ suffices to establish our claim. By the internality of this conclusion, provisionally assume standardness of $x, y \in M$ as well as of $g \in G$. Choose some $g'$ with $\imath(g', g)$ and any standard $\varepsilon > 0$. We have $d(g \circlearrowright' x, g' \circlearrowright x) < \frac{\varepsilon}{2}$ and similarly for $y$ by the defining property of the function $\circlearrowright'$.

As usual, we repeatedly apply the triangle inequality to obtain

$$\begin{aligned} d(x,y) &= d(g'\circlearrowright x, g'\circlearrowright y) \\ &\le d(g\circlearrowright' x, g'\circlearrowright y) + d(g'\circlearrowright x, g\circlearrowright' x) \\ &\le d(g\circlearrowright' x, g'\circlearrowright y) + \frac{\varepsilon}{2} \\ &\le d(g\circlearrowright' x, g\circlearrowright' y) + d(g'\circlearrowright y, g\circlearrowright' y) + \frac{\varepsilon}{2} \\ &\le d(g\circlearrowright' x, g\circlearrowright' y) + \frac{\varepsilon}{2} + \frac{\varepsilon}{2} \\ &= d(g\circlearrowright' x, g\circlearrowright' y) + \varepsilon. \end{aligned}$$

The inequality $d(x,y) \le d(g\circlearrowright' x, g\circlearrowright' y) + \varepsilon$ holds for any standard $\varepsilon$. Using a short Transfer argument we get that $d(x,y) \le d(g\circlearrowright' x, g\circlearrowright' y)$ also holds. Discharging the standardness assumptions, the conclusion holds for all $x, y \in M$ and all $g \in G$, and so the periodic subgroups act by isometries.
**Qed.**

**2.3.11.** Theorem 2.3.9 places limitations on groups which are approximated by non-standard groups that act on standard manifolds, especially for the approximation of periodic groups. We have of course that dihedral groups admit isometric actions on the circle $S^1$, which severely constrains groups with dihedral approximations. Similarly, groups of the form $\mathbb{Z}/n\mathbb{Z} \times \mathbb{Z}/m\mathbb{Z} : \mathbb{Z}/\ell\mathbb{Z}$ for $\ell \in \{2,3,4,6\}$ are the ones that admit $K$-Lipschitz actions on the torus $S^1 \times S^1$ [2].

**2.3.12. Theorem** (Manevitz-Weinberger, [30]-Theorem 1)**.** Fix some positive $K \in \mathbb{R}$. Consider a compact manifold $M$ which has a faithful $K$-Lipschitz $\mathbb{Z}/n\mathbb{Z}$ action for all $n \in \mathbb{N}$. Then $M$ has a faithful $K$-Lipschitz action by $\mathbb{Q}/\mathbb{Z}$.

**Proof.** Provisionally assume $\operatorname{st}(K)$ and $\operatorname{st}(M)$. Using the locally finiteness of $\mathbb{Q}/\mathbb{Z}$, we can apply Proposition 2.2.36 to obtain an approximation $\iota$ of $\mathbb{Q}/\mathbb{Z}$ via some finite subgroup $H < \mathbb{Q}/\mathbb{Z}$. Such a subgroup must have the form $\mathbb{Z}/\omega\mathbb{Z}$ for some $\omega \in \mathbb{N}$, and therefore $H$ admits a faithful $K$-Lipschitz action on $M$. Applying Theorem 2.3.9 we get a faithful $K$-Lipschitz action on $M$ by $\mathbb{Q}/\mathbb{Z}$.
**Qed.**

**2.3.13.** It is natural to ask whether the $K$-Lipschitz assumption of Theorem 2.3.9 can be replaced with some weaker condition. This remains to be seen. Clearly, any potential condition would have to imply the continuity and S-continuity of the action. However, neither continuity nor S-continuity suffice as a replacement: the proof relies on both conditions, and since the action $\circlearrowleft : H \times M \to M$ is not standard, it might satisfy one continuity condition but not the other. Moreover, the fact that the same condition appears in both the assumptions and the conclusion makes it hard to find plausible candidates[1].

## 2.4 Snappy groups

**2.4.1.** Motivated by the negative result on ordinary approximation of $SO(3)$ mentioned in 2.1.8, one wishes to say something about the existence of well-behaved approximations of $SO(3)$ in the new formalism.

**2.4.2. Proposition.** One cannot approximate the group $SO(3)$ using any of its finite subgroups.

**Proof.** Evidently one cannot approximate $SO(3)$ by a standard finite subgroup. So assume that the predicate $\iota$ approximates $SO(3)$ via some nonstandard finite subgroup $H < SO(3)$. By the classification of finite subgroups of $SO(3)$, every subgroup of $SO(3)$ of order $> 60$ arises as a dihedral group, so we can assume $H = D_\omega$ for some non-standard $\omega \in \mathbb{N}$. Since $D_\omega$ admits a continuous isometric action on the circle, so does every periodic subgroup of $SO(3)$. But $A_5 < SO(3)$ admits no such action, a contradiction. Hence one cannot approximate $SO(3)$ using finite subgroups.
**Qed.**

**2.4.3.** We can use Proposition 2.4.2 as a stepping stone for obtaining further results on non-approximability. For example, by considering a special property (snappiness, Definition 2.4.5) of the group $SO(3)$, we get that (unlike the approximations which we obtain for, say, profinite groups), the approximations of $SO(3)$ never respect the usual topology of the group.

**2.4.4. Definition.** Consider a group $H$ and a standard topological group $G$ represented as an equivalence space $(G, \multimapinv)$. We call a function $f : H \to G$ an *S-near-homomorphism* if $1_G \multimapinv f(1_H)$, and for any $x, y \in H$ we have $f(xy) \multimapinv f(x)f(y)$.

**2.4.5. Definition.** We call a standard topological group $G$ represented as an equivalence space $(G, \multimapinv)$ *snappy* if for any finite group $H$ and S-near-homomorphism $f' : H \to G$

[1] That said, moduli-of-continuity conditions do deserve further investigation!

we can find a group homomorphism $f : H \to G$ such that for all $x \in H$, $f(x) \multimapinv f'(x)$.

**2.4.6.** The term *snappy* of Definition 2.4.5 intends to evoke a picture of a DIP socket: given a light jiggle, the chip's electrical connecting pins gently snap into place. In the same way, giving a gentle jiggle to a near-homomorphism makes all the points that just barely missed their holes snap into place.

**2.4.7.** We prove the snappiness of $SO(3)$ in Corollary 2.4.9. A result of Babai, Friedl and Lukács [3] perfectly encapsulates the group-theoretic part of the argument, so we only have to do the non-standard analytic reasoning.

**2.4.8. Theorem** (Babai-Friedl-Lukács)**.** Let $|-|$ denote the Euclidean matrix norm on $SO(3)$. Fix a positive $\varepsilon < 0.001$ and a finite group $H$. Consider a function $f' : H \to SO(3)$ such that $|I - f'(1)| < \varepsilon$ and $|f'(x)f'(y) - f'(xy)| < \varepsilon$ for all $x, y \in H$ as well. Then we can find a group homomorphism $f : H \to SO(3)$ such that for all $x \in H$ the inequality $|f(x) - f'(x)| < 1000\varepsilon$ holds.

**Proof.** See the proof of [3]-Theorem 1.3.
**Qed.**

**2.4.9. Corollary.** The group $SO(3)$ is snappy.

**Proof.** The equivalence of all matrix norms guarantees that (in accordance with Theorem 1.3.35) the binary predicate $M_1 \multimapinv M_2$ defined as $\forall^{st}\varepsilon > 0.|M_1 - M_2| < \varepsilon$ universally represents the topology of $SO(3)$ as an equivalence space. In particular we have the S-continuity of the group operations with respect to this relation. Consider any S-near-homomorphism $f' : H \to SO(3)$. Take a standard finite set $S \subseteq (0, 0.001)$. Denote $s = \min S$. We have st $\left(\frac{s}{1000}\right)$ by Corollary 1.2.11, so the inequality $|f'(x)f'(y) - f'(xy)| < \frac{s}{1000}$ obtains for all $x, y \in H$. Theorem 2.4.8 applies and gives us a group homomorphism $f : H \to SO(3)$ such that for all $x \in H$, $|f(x) - f'(x)| < s$. This proves that for any standard finite set of numbers we can find a group homomorphism $f$ such that for all $\varepsilon \in S$, $|f(x) - f'(x)| < \varepsilon$. By the principle of Idealization, we conclude the existence of a group homomorphism $f : G \to SO(3)$ such that $\forall^{st}\varepsilon > 0.\forall x \in H.|f(x) - f'(x)| < \varepsilon$. Consequently, $\forall x \in H.f(x) \multimapinv f'(x)$, which proves the snappiness of the group $SO(3)$.
**Qed.**

**2.4.10. Theorem.** $SO(3)$ does not admit internal, robust finite approximations for its usual topology.

**Proof.** Assume for a contradiction that we have a finite group $H$ and the required internal approximation predicate $\iota$ relating elements of $H$ to elements of $SO(3)$. This means that the following hold:

A1 For any standard $g \in SO(3)$ we can find $h \in H$ such that $\iota(h, g)$ holds.

A2 For any standard $g \in SO(3)$, $h_1, h_2 \in H$ such that $\iota(h_1, g)$ and $\iota(h_2, g)$ hold, we have $h_1 = h_2$.

T1 For any standard $g_1, g_2 \in SO(3)$ and any $h \in H$ such that $\iota(h, g_1)$ and $\iota(h, g_2)$ both hold, we have $g_1 \multimapinv g_2$.

A4 For any $g_1, g_2 \in SO(3)$ and any $h_1, h_2 \in H$ with $\iota(h_1, g_1)$ and $\iota(h_2, g_2)$ we have $\iota(h_1 h_2, g_1 g_2)$.

Without loss of generality, we can assume $H = \overline{H} = \{x \in H \mid \exists g \in SO(3).\iota(h, g)\}$: if $\iota$ approximates $SO(3)$ via $H$, then it does the same via $\overline{H}$. Consider the set defined by $E = \{(h, g) \in H \times SO(3) \mid \iota(h, g)\}$. The set $E$ may not form the graph of a function: we cannot rule out having $\iota(h, g_1)$ and $\iota(h, g_2)$ for non-standard $g_1, g_2 \in G$. However, $H = \overline{H}$, so $\forall h \in H.\exists g \in G.\iota(h, g)$, and consequently we can apply the Axiom of Choice to get a function $e' : H \to SO(3)$ that satisfies $\iota(h, e'(h))$ for all $h \in H$. We claim that $e' : H \to SO(3)$ gives an S-near-homomorphism between $H$ and $SO(3)$. We must show that $e'(h_1)e'(h_2) \multimapinv e'(h_1 h_2)$ for all $h_1, h_2 \in H$. We have $\iota(h_1, e'(h_1))$ and $\iota(h_2, e'(h_2))$. By [A4] we get $\iota(h_1 h_2, e'(h_1)e'(h_2))$. But we also have $\iota(h_1 h_2, e'(h_1 h_2))$, so by [T1] $e'(h_1)e'(h_2) \multimapinv e'(h_1 h_2)$ as we desired. We know from Corollary 2.4.9 that $SO(3)$ forms a snappy group: we deduce the existence of a group homomorphism $e : H \to SO(3)$ such that $\forall h \in H.e(h) \multimapinv e'(h)$. The image $e(H)$ necessarily forms a finite subgroup of $SO(3)$. We prove that $e(H)$ approximates $SO(3)$. Define the approximation predicate $\xi(x, g)$ between $x$ and $g$ as an abbreviation for $\exists h \in H.x = e(h) \wedge \iota(h, g)$. We have to prove four things:

1. For any standard $g \in SO(3)$ we can find $x \in e(H)$ such that $\xi(x, g)$ holds. Start by using [A1] to find $h \in H$ with $\iota(h, g)$. Set $x = e(h)$.

2. For any standard $g \in SO(3)$ and $x_1, x_2 \in e(H)$ such that $\xi(x_1, g)$ and $\xi(x_2, g)$ both hold, we have $x_1 = x_2$. By $\xi(x_1, g)$ we have some $h_1$ such that $x_1 = e(h_1)$ and

$\iota(h_1, g)$. By $\xi(x_2, g)$ we have some $h_2$ such that $x_2 = e(h_2)$ and $\iota(h_2, g)$. Since we have $\iota(h_1, g)$ and $\iota(h_2, g)$ we can apply [A2] to get $h_1 = h_2$. But then $x_1 = e(h_1) = e(h_2) = x_2$ as required.

3. For any standard $g_1, g_2 \in SO(3)$ and $x \in e(H)$ such that $\xi(x, g_1)$ and $\xi(x, g_2)$ both hold, we have $g_1 = g_2$. We have some $h_1$ such that $x = e(h_1)$ and $\iota(h_1, g_1)$ holds. We also have some $h_2$ such that $x = e(h_2)$ and $\iota(h_2, g_2)$ holds. We also have $\iota(h_1, e'(h_1))$ and $\iota(h_2, e'(h_2))$ by definition of the function $e'$. Hence [T1] gives us $g_1 \multimapinv e'(h_1)$ and $g_2 \multimapinv e'(h_2)$. But then we have the following chain of nearness relationships: $g_1 \multimapinv e'(h_1) \multimapinv e(h_1) = x = e(h_2) \multimapinv e'(h_2) \multimapinv g_2$, so $g_1 \multimapinv g_2$. Since we have $\mathrm{st}(g_1)$ and $\mathrm{st}(g_2)$, the Fréchet property (Definition 1.3.25) guarantees $g_1 = g_2$.

4. For any standard $g_1, g_2 \in SO(3)$, $x_1, x_2 \in e(H)$ such that $\xi(x_1, g_1)$ and $\xi(x_2, g_2)$, we have $\xi(x_1x_2, g_1g_2)$. By $\xi(x_1, g_1)$ we have some $h_1$ such that $x_1 = e(h_1)$ and $\iota(h_1, g_1)$. By $\xi(x_2, g_2)$ we have some $h_2$ such that $x_2 = e(h_2)$ and $\iota(h_2, g_2)$. We need to construct some $h \in H$ such that $x_1x_2 = e(h)$ and $\iota(h, g_1g_2)$. But we have $\iota(h_1h_2, g_1g_2)$ using [A4]. Set $h = h_1h_2$. We have $e(h) = e(h_1h_2) = e(h_1)e(h_2) = x_1x_2$.

The finite subgroup $e(H)$ of $SO(3)$ approximates $SO(3)$ internally, contradicting Proposition 2.4.2.

**Qed.**

**2.4.11. Problem.** Can one extend the proof of Theorem 2.4.10 to non-robust approximations by proving a more general version of Theorem 2.4.8?

# Chapter 3

# Other results

In this short chapter we present some of our results that do not concern structural approximation directly, but relate to the development of algebra in Internal Set Theory.

## 3.1 Monotone subsequences

**3.1.1.** Baszczyk, Kanovei, Katz and Nowik [5] have recently presented an ultrapower proof of the following classical result of Real Analysis: every infinite sequence in a totally ordered set contains either an infinite constant subsequence or an infinite strictly monotone subsequence. Inspired by their argument, we give a straightforward, ultrapower-free proof using Internal Set Theory.

**3.1.2. Theorem.** Every infinite sequence in a totally ordered set contains either an infinite constant subsequence or an infinite strictly monotone subsequence.

**Proof.** Consider a totally ordered set $(S, <)$, and a sequence $a : \mathbb{N} \to S$. We can provisionally assume the standardness of both the set $S$ and the sequence $a$. Take any non-standard $\omega \in \mathbb{N}$. Define the following sets:

$$A_1 = \{\!\!\{ k \in \mathbb{N} \,\big|\, a_k < a_\omega \}\!\!\}$$
$$A_2 = \{\!\!\{ k \in \mathbb{N} \,\big|\, a_k = a_\omega \}\!\!\}$$
$$A_3 = \{\!\!\{ k \in \mathbb{N} \,\big|\, a_k > a_\omega \}\!\!\}$$

The Standardization axiom ensures the standardness of all three sets $A_1, A_2, A_3 \subseteq \mathbb{N}$. It follows by Corollary 1.2.10 that we have $\mathrm{st}(A_1 \cup A_2 \cup A_3)$. Any standard natural number $n \in \mathbb{N}$ satisfies one of the sentences $a_n < a_\omega$, $a_n = a_\omega$ or $a_n > a_\omega$ and so $A_1 \cup A_2 \cup A_3$

contains every standard natural. By Transfer we immediately get $A_1 \cup A_2 \cup A_3 = \mathbb{N}$. Consider the following:

1. If $\forall i \in A_1.\exists m \in A_1.i < m \wedge a_i < a_m$ holds, then we can construct an infinite, monotone increasing subsequence of $a$ by taking indices in $A_1$.

2. If $\forall j \in A_3.\exists n \in A_3.i < n \wedge a_j > a_n$ holds, then we can construct an infinite, monotone decreasing subsequence of $a$ by taking indices in $A_3$.

However, if the two previous conditions both fail, then

1. We can find $i \in A_1$ such that for all $m \in A_1$, if $i < m$ then $a_i \geq a_m$.

2. We can find $j \in A_3$ such that for all $n \in A_3$, if $j < n$ then $a_j \leq a_n$.

By Transfer, we can choose standard values for both $i$ and $j$; this means that we have $i < \omega$ and $j < \omega$. Assume for a contradiction that $\omega \in A_1$. Then we have $a_i \geq a_\omega$. However, $\mathrm{st}(i)$ and $i \in A_1$ both hold, so by definition $a_i < a_\omega$, a contradiction. Therefore, $\omega \notin A_1$. Similarly, assume that $\omega \in A_3$. Then we have $a_j \leq a_\omega$. However, $\mathrm{st}(j)$ and $j \in A_3$ both hold, so by definition $a_j > a_\omega$, a contradiction. Therefore, $\omega \notin A_3$.
Since $\omega \in \mathbb{N} = A_1 \cup A_2 \cup A_3$, we must then have $\omega \in A_2$. A standard finite set has all its elements standard (Theorem 1.2.5), but $\omega$ is not standard, so $A_2$ is not finite. But we have $\forall^{st} n, m \in A_2.a_n = a_m$. By Transfer the sequence $a$ is constant on the infinite set $A_2 \subseteq \mathbb{N}$, so $a$ has an infinite constant subsequence.
**Qed.**

**3.1.3.** The usual proofs of the monotone subsequence theorem go through the Bolzano-Weierstrass theorem: a bounded sequence has a convergent subsequence, and a convergent sequence has a constant or monotone subsequence; similarly, unbounded sequences always have a monotone divergent subsequence. The ultrapower proof of Baszczyk, Kanovei, Katz and Nowik [5] and the Internal Set Theory proof presented above both bypass the convergence considerations, and so they work without modification in any ordered structure (see also [5]-Remark 3.4. for the relation between the ultrapower proof and proofs based on the idea of *peaks*).

## 3.2 Sheaves

### Motivation

**3.2.1.** The functional interpretation [46] of non-standard arithmetic **P** (Peano arithmetic in finite types with the Axiom of Extensionality, the Idealization axiom and the Herbrandized Axiom of Choice; a weak subsystem of Nelson's Internal Set Theory) allows one to extract a finite list $t(x)$ from each **P**-proof of $\forall^{st} x.\exists^{st} y.\varphi(x,y)$, in such a way that Peano arithmetic in finite types itself (without the additional non-standard axioms) proves $\forall x.\exists y \in t(x).\varphi(x,y)$. Observing that one can bring the non-standard definitions of continuity (Definition 1.3.15), compactness (Theorem 1.3.33), Riemann-integrability etcinto Nelson normal form in **P**, Sanders [41] formulated a technique ($\mathfrak{C}\mathfrak{I}$) that allows one to convert theorems formulated purely[1] in terms of these non-standard definitions into associated theorems that have effective computational content and no longer involve non-standard notions.

**3.2.2.** Techniques such as $\mathfrak{C}\mathfrak{I}$ have seen successful applications in constructive analysis, topology and measure theory. To one day apply similar techniques in algebra, one needs to find equivalences between nonstandard and classical algebraic notions. Since we already have a large library of such equivalences in analysis and topology, it's natural to start looking for new equivalences where these fields intersect algebra, e.g. in sheaf theory.

**3.2.3.** Here we give a pure non-standard characterization of sheaves on topological spaces. Apart from placing sheaves in the domain of applicability of $\mathfrak{C}\mathfrak{I}$-style techniques, our characterization also realizes directly a conceptual view of sheaves as "continuous set-valued maps" enunciated by Vickers [47] which cannot be formalized by topologizing the class of sets in the ordinary way.

### Predicated quivers

**3.2.4.** One can regard the predicated spaces of Definition 1.3.14 as (external) reflexive graphs. Generalizing to external reflexive *quivers* offers a very quick, intuitive path to defining sheaves on topological spaces.

[1] Quite literally: the term extraction algorithm underlying the technique ignores internal axioms, so one has to express all the hypotheses and the conclusion in terms of the non-standard notions.

**3.2.5. Definition.** A *predicated quiver* consists of the following data:

- an *underlying edge set* $E$,
- an *underlying vertex set* $T$,
- a *reflexivity map* $r : T \to E$
- a ternary predicate in the language of Internal Set Theory, $e : x \multimapinv y$, with $e$ ranging over the set of edges $E$, and $x, y$ ranging over the set of vertices $T$,

subject to the following conditions:

- for each $x \in T$, $r(x) : x \multimapinv x$, and
- if $e : x \multimapinv y$ and $e : x' \multimapinv y'$ then $x = x'$ and $y = y'$.

**3.2.6.** One can see every predicated space as a predicated quiver by setting $E = T^2$, taking $r$ as the diagonal map $T \to T^2$ and defining $(a,b) : x \multimapinv y$ precisely if $a = x$, $b = y$ and $a \multimapinv b$. Similarly, one can see any small category as a predicated quiver by treating its objects as vertices, its morphisms as edges, and taking $r$ as the map that sends each object to its identity morphism. We can define maps between predicated quivers analogously to how we defined continuous maps between predicated spaces.

**3.2.7. Definition.** An *S-continuous map between two predicated quivers* $(E_1, T_1, r_1)$ and $(E_2, T_2, r_2)$ consists of a pair of functions $f_E : E_1 \to E_2$ and $f_T : T_1 \to T_2$ subject to the following conditions:

- for every standard $x \in T$, every $y \in T$ and every $e : x \multimapinv y$ we have $f_E(e) : f_T(x) \multimapinv f_T(y)$, and
- for every $x \in T$, we have $f_E(r_1(x)) = r_2(x)$.

**3.2.8.** The fact that predicated quivers treat both topological spaces and categories on an equal footing allows us to define a very simple and well-behaved sheaf-like notion over a predicated quiver: an S-continuous map from the predicated quiver to the category $\mathfrak{Set}$, itself seen as a predicated quiver (modulo size issues, which one can treat easily in this case, e.g. via the Replacement axiom). We show that in the topological case this construction gives rise to actual sheaves.

## Predicated Sheaves

**3.2.9. Definition.** We call an S-continuous map from a predicated quiver $(E, T, r)$ to a small subcategory of the category of sets (regarded as a predicated quiver) a *predicated sheaf* on $(E, T, r)$.

**3.2.10.** When we consider a predicated sheaf $\varphi = (f_E, f_T)$ on the predicated space $(T, \multimap)$, we denote the vertex map $f_T(x)$ as $\varphi_x$, and for $x \multimap y$ the edge map $f_E((x, y))$ as $\varphi_x^y$.

**3.2.11. Definition.** Consider a predicated sheaf $\varphi$ over the space $(T, \multimap)$. We call a function $f : (x \in T) \to \varphi_x$ an *S-section* of $\varphi$ over the S-open set $U \subseteq T$ if for all standard $x \in U$ and arbitrary $y \in U$ satisfying $x \multimap y$, we have $\varphi_x^y(f(x)) = f(y)$.

**3.2.12.** Recall that a predicated space $(T, \multimap)$ represents a topological space $(T, \Omega T)$ if the standard S-open sets of $(T, \multimap)$ coincide with the standard open sets of $(T, \Omega T)$. Similarly, a predicated sheaf represents a sheaf if its standard S-sections coincide with the sections of the represented sheaf.

**3.2.13. Definition.** Take a standard topological space $T$ equipped with a standard sheaf $\mathcal{F}$. We say that the predicated sheaf $\varphi$ over $T$ *represents the sheaf* $\mathcal{F}$ if the following hold:

- $\mathcal{F}_x = \varphi_x$ for all $x \in T$, where $\mathcal{F}_x$ denotes the stalk of $\mathcal{F}$ at point $x$;
- for every standard open $U$ and section $f \in \mathcal{U}$, the map $x \mapsto [f]_x$ forms an S-section of $\varphi$; and
- for every standard open $U$ and S-section $\overline{f} : (x \in U) \to \varphi_x$ we can find a section $f \in \mathcal{F}(U)$ such that $\forall x \in U . \overline{f}(x) = [f]_x$,

where $[f]_x$ denotes the sheaf-theoretic germ of the section $f$ at the point $x$.

**3.2.14. Lemma.** Consider a topological predicated space $(T, \multimap)$, and a standard point $p \in T$. We can find an open set $P \ni p$ that consists entirely of points near $p$, i.e. such that for all $y \in P$ we have $p \multimap y$.

**Proof.** Take a topological predicated space $(T, \multimap)$, and a standard point $p \in T$. Consider any standard finite set $\mathcal{U}$ of open sets containing $p$. The finite intersection $P =$

$\bigcap \mathcal{U}$ contains $p$, and lies inside every open $U \in \mathcal{U}$. By the Idealization axiom, we obtain an open set $P$ containing $p$ that lies inside every standard open $U \in \Omega T$ containing the point $p$. Pick any standard open $N$ with $p \in N$, and consider any $y \in P$. We have $P \subseteq N$, so $y \in N$. This proves that $p \multimapinv y$.
**Qed.**

**3.2.15. Theorem.** Consider a standard sheaf $\mathcal{F}$ on a standard topological space $T$. We can find a predicated sheaf $\varphi$ that represents $\mathcal{F}$ in the sense that for every standard section $f$ of $\mathcal{F}(U)$ the function $x \mapsto [f]_x$ yields an S-section, and for standard S-section $q$ of $\varphi$ over $U$ we can find a function $f$ such that $\forall x \in U.q(x) = [f]_x$.

**Proof.** Consider a standard sheaf $\mathcal{F}$ on a standard topological space $T$. We can find a standard map $c$ that assigns to each germ $a$ around $x$ a representative $c(x,a) = (V,s)$ such that $V$ forms an open neighborhood of $x$, $s \in \mathcal{F}(V)$ and $[c(x,a)]_x = a$. Set $\varphi_x = \mathcal{F}_x$ and $\varphi_x^y(a) = [c(x,a)]_y$ (define the value of the function however you wish for pairs $(x,y) \in T^2$ with $\neg x \multimapinv y$).
First take a standard section $f \in \mathcal{F}(U)$. Notice that we get $\mathrm{st}(U)$ automatically. We want to prove that the map $x \mapsto [f]_x$ forms an S-section of the predicated sheaf $\varphi$ over $U$. According to Definition 3.2.11, this happens precisely if for all standard $x \in U$ and arbitrary $y \in U$ with $x \multimapinv y$, we have $\varphi_x^y\left([f]_x\right) = [f]_y$. Substituting the definition of $\varphi_x^y$ given above, we want $\left[c\left(x,[f]_x\right)\right]_y = [f]_y$. Since we have $\mathrm{st}(f), \mathrm{st}(x)$ and $\mathrm{st}(c)$, Corollary 1.2.11 guarantees the standardness of the section $c\left(x,[f]_x\right)$. Since $\left[c\left(x,[f]_x\right)\right]_x = [f]_x$, we can find a standard open $X$ around $x$ on which $c\left(x,[f]_x\right)|_X = f|_X$. But $x \multimapinv y$, so $y \in X$. Since $c\left(x,[f]_x\right)$ and $f$ agree on a neighborhood of $y$, they have the same germ at $y$, i.e. $\left[c\left(x,[f]_x\right)\right]_y = [f]_y$. Therefore, the map $x \mapsto [f]_x$ forms an S-section of $\varphi$ over $U$. This takes care of one direction.
Now consider a standard S-section $q : (x \in U) \to \mathcal{F}_x$. We want to find a section $f \in \mathcal{F}(U)$ such that $\forall x \in U.[f]_x = q_x$. We divide this long proof into several claims. If we have some open set $M \subseteq U$ and section $s \in \mathcal{F}(M)$ such that $\forall m \in M.[s]_m = q_m$, then we call $(M,s)$ a *partial solution*. Notice that the predicate "$(M,s)$ constitutes a partial solution" has no non-standard parameters, so Transfer applies to it.

**Claim 1**: A partial solution exists around every standard point $x \in U$.

Take a standard point $x \in U$. By the S-section condition, we know the following:

$$\forall y \in U.x \multimapinv y \to [c(x,q_x)]_y = q_y$$

Using Lemma 3.2.14, we can pick an open set $X$ containing $x$ such that we have $\forall y.y \in X \to x \multimap y$. We know that $c(x, q_x)$ is a standard section defined on a standard open containing $x$. But $X$ lies inside every standard open that contains $x$, so we can restrict $c(x, q_x)$ to $X$. By setting $s = c(x, q_x)|_X \in \mathcal{F}(X)$, the S-section condition yields

$$\forall y \in X.[s]_y = q_y$$

and so $(X, s)$ constitues a partial solution. This proves our claim.

**Claim 2**: A standard maximal partial solution exists.

Assume that we have a non-empty $I$-indexed chain $(M_i, s_i)$ of partial solutions. The union $M = \bigcup_{i \in I} M_i$ still is itself an open set, and the $M_i$ form an open cover of $M$. We know that if $i < j$ then $[s_i]_m = q_m = [s_j]_m$ for each $m \in M_i$, so by the locality condition $s_i = s_j|_{M_i}$. By the gluing condition, we get a section $s \in \mathcal{F}(M)$, and for each $m \in M$ we have some $i$ such that $m \in M_i$, and there $[s]_m = [s_i]_m = q_m$. Thus $(M, s)$ gives an upper bound of the chain. By Zorn's lemma, a maximal partial solution exists. By Transfer, a standard maximal partial solution exists.

**Claim 3**: Standard maximal partial solutions $(M, s)$ have $M = U$.

Assume for a contradiction that we have a standard maximal partial solution $(M, s)$ with $M \subsetneq U$. This means that we find a point $p \in U$ such that $p \notin M$. By Transfer, we can assume $\mathrm{st}(p)$. We will extend the partial solution to this standard point $p$.
By Claim 1, we can find a partial solution $(P, t)$ with $P \ni p$. Thus we have $\forall m \in M.[s]_m = q_m$ and $\forall y \in P.[s]_y = q_y$. This means that $\forall y \in P \cap M.[s]_y = [t]_y$, so by the locality condition $s|_{P \cap M} = t|_{P \cap M}$. Since $s$ and $t$ agree on the intersection of their domains of definition, we can glue them together to obtain a partial solution containing both $M$ and $p$, and contradicting the maximality of $M$.
**Qed.**

**3.2.16.** From Theorem 3.2.15 we know that we can represent any standard sheaf on a standard topological space using a predicated sheaf. We prove the converse as well.

**3.2.17. Proposition.** Every standard predicated sheaf on a topological predicated space $T$ represents some standard sheaf.

**Proof.** Consider the standard predicated sheaf $\varphi$ on the standard space $T$. Write $\Phi$ for the standard set $\bigcup_{x\in T}\varphi_x$. First for any *standard* open set $U \in \Omega T$ define

$$\mathcal{F}[U] = \\ \left\{\!\!\left| f \in \mathcal{P}(T\times\Phi) \,\middle|\, \left(f : (z\in U)\to\varphi_z\right) \wedge \forall^{st} x.\forall y.x \multimap y \to \varphi_x^y(f(x)) = f(y) \right|\!\!\right\}.$$

Now we can define our sheaf $\mathcal{F}$ as

$$\mathcal{F} = \{\!|(U,F)\in\Omega T\times\mathcal{P}(\mathcal{P}(T\times\Phi)) \mid F = \mathcal{F}[U]|\!\}.$$

For any standard $U\in\Omega T$ we can find exactly one standard set $F$ such that $(U,F)\in\mathcal{F}$. By Transfer, we get that $\mathcal{F}$ is a function. Similarly, for any standard $U\in\Omega T$ the value $\mathcal{F}(U)$ forms a subspace of the function space $(x\in U)\to\varphi_x$, and Transfer ensures that the same holds for arbitrary $U$. Therefore, we can define our restriction maps $\mathrm{res}_U^V : \mathcal{F}(V)\to\mathcal{F}(U)$ the obvious way, as restrictions of functions. To see that this map is well-defined, just apply Transfer to the formula

$$\forall^{st} U\in\Omega T.\forall^{st} V\in\Omega T.U\subseteq V\to\forall^{st} f\in\mathcal{F}(V).\mathrm{res}_U^V(f)\in\mathcal{F}(U).$$

To prove locality, take an open set $U$, an $I$-indexed open cover $U_i$ of $U$, and two sections $s,t\in\mathcal{F}(U)$. To show that $s=t$, it suffices to show that for all $x\in U$ we have $s(x)=t(x)$. But each $x\in U$ belongs to some $U_i$, and we see $s(x)=\mathrm{res}_{U_i}^U(s)(x)=\mathrm{res}_{U_i}^U(t)(x)=t(x)$. To prove gluing, take an open set $U$, an $I$-indexed open cover $U_i$ of $U$ and an $I$-indexed set of sections $s_i\in\mathcal{F}(U_i)$. Provisionally assume the standardness of all these objects. For each $x\in X$ pick an $i$ such that $U_i$ covers $x$, and define the function

$$\begin{aligned} &s : (x\in U)\to\varphi_x \\ &s(x) = s_i(x) \end{aligned}$$

Since the sections $s_i$ agree on all intersections of the cover, the function $s$ is well-defined and standard. We need to prove that $s\in\mathcal{F}(U)$, which happens precisely if $s$ satisfies $\forall^{st} x\in U.\forall y\in U.x\multimap y\to\varphi_x^y(s(x))=s(y)$. So take any standard $x\in U$. We can find some $i\in I$ such that $U_i$ covers $x$. By Transfer we can pick such $i$ (and hence $U_i$ and $s_i$) as standard. Hence, by $x\multimap y$ we get that $y\in U_i$. Moreover, by the standardness of $s_i\in\mathcal{F}(U_i)$, we get that $\varphi_x^y(s_i(x))=s_i(y)$. But $s_i(x)=s(x)$ and $s_i(y)=s(y)$, so $\varphi_x^y(s(x))=s(y)$, and so $s\in\mathcal{F}(U)$ as desired. Transfer gets rid of the provisional assumptions, and we conclude that $\mathcal{F}$ forms a sheaf over the space $T$.

**Qed.**

## The Alexandroff case

**3.2.18.** Over an Alexandroff space $(T, \leq)$, Theorem 3.2.15 acquires a much stronger form: every sheaf, whether standard or non-standard, corresponds to a predicated sheaf.

**3.2.19.** To a point $a \in T$ of the Alexandroff space, we can associate the upper set $\uparrow a = \{x \in T \mid a \leq x\}$, and $\uparrow a$ forms the smallest open set containing $a$. Given a sheaf $\mathcal{F}$ over $T$ and sections $s, t \in \mathcal{F}(U)$, we have $[s]_a = [t]_a$ precisely if $s|_{\uparrow a} = t|_{\uparrow a}$. Hence we can identify the stalk $\mathcal{F}_a$ with the set of local sections $\mathcal{F}(\uparrow a)$.

**3.2.20.** By contravariance, whenever we have $a \leq b$, we have $\uparrow b \subseteq \uparrow a$. Functoriality of $\mathcal{F}$ gives a map $\mathcal{F}_a^b : \mathcal{F}(\uparrow a) \to \mathcal{F}(\uparrow b)$. Since we identify stalks $\mathcal{F}_a$ with sets of local sections $\mathcal{F}(\uparrow a)$, we can see $\mathcal{F}_a^b$ as a map $\mathcal{F}_a^b : \mathcal{F}_a \to \mathcal{F}_b$, corresponding directly to the map $\varphi_a^b$ of Theorem 3.2.15.

## Local definition

**3.2.21.** The correspondence between sheaves and predicated sheaves allow us to define sheaves in a local, pointwise fashion. This approach works very well for sheaves whose local behavior is easier to specify than its global behavior. Water flow in a network of pipes provides a salient example. Locally, we only have one constraint: the amount of water flowing into a point should equal the amount of water flowing out from that point. However, specifying a global flow requires understanding the topology of the entire pipe network.

**3.2.22.** As a simple example, consider the three-way junction $Y$ of pipes depicted on Figure 3.1. Treat the network $Y$ as a subspace of $\mathbb{R}^2$ equipped with the usual Euclidean topology. We want to define a sheaf whose global sections correspond to possible flows on the network. We define a predicated sheaf as an S-continuous map of quivers $\varphi$. We begin with the vertex map (the stalks) as follows:

$$\varphi_x = \begin{cases} \left\{(l, ur, ul) \in \mathbb{R}^3 \,\middle|\, l + ur + ul = 0\right\} & \text{if } x = p \\ \left\{(l, r) \in \mathbb{R}^2 \,\middle|\, l + r = 0\right\} & \text{otherwise} \end{cases}$$

We can define the transition maps by cases as well:

$$\varphi_x^y = \begin{cases} (l, ur, ul) \mapsto (-ur, ur) & \text{if } x = p \text{ and } y \in UR \\ (l, ur, ul) \mapsto (-ul, ul) & \text{if } x = p \text{ and } y \in UL \\ (l, ur, ul) \mapsto (-l, l) & \text{if } x = p \text{ and } y \in L \\ (-a, a) \mapsto (-a, a) & \text{otherwise.} \end{cases}$$

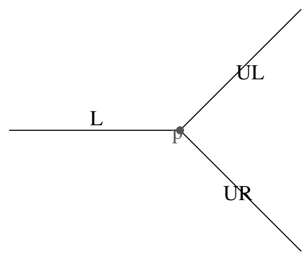


Figure 3.1: A three-way junction on a network.

**3.2.23. Problem.** Consider a small category $\mathfrak{C}$ equipped with a coverage or Grothendieck topology. Can we turn $\mathfrak{C}$ into a predicated quiver in such a way that sheaves on $\mathfrak{C}$ correspond to sheaves in the sense of Grothendieck?

# Chapter 4

# Mechanization

## 4.1 Computer-verified proofs

**4.1.1.** In 1998, Simpson [44] gave a counterexample to the Homotopy Hypothesis, contradicting an earlier, widely accepted argument by Kapranov and Voevodsky [27]. It took until 2013 for Voevodsky to track down the error in his own argument. This long struggle led Voevodsky to formulate his Univalent Foundations program [48], which ultimately led to significant advances in the computer verification (mechanization) of proofs, including the development of the field of research now known as Homotopy Type Theory [45].

**4.1.2.** Unfortunately, our more modest field appears no less vulnerable to erroneous arguments than algebraic topology. The main result of Chapter 2, Theorem 2.3.9, had a direct precursor in the form of the Manevitz-Weinberger theorem on discrete circle actions (Theorem 2.3.12). However, the original proof of that result (see [30]-Theorem 1) contained a significant mistake that went unnoticed until Imamura [25] found and fixed the problem a full twelve years later.

**4.1.3.** The warnings of Sections 1.1.31 and 1.2.13 suggest that Internal Set Theory (and to some extent model-theoretic non-standard analysis) might have unusual susceptibility to accidental mistakes due to its reliance on syntactic restrictions on set formation and induction principles that have universal validity in the rest of mathematics. Given the history of Theorem 2.3.12, we decided to computer-verify our proof of Theorem 2.3.9 by formalizing the argument in an extension of Martin-Löf Type Theory, and checking the correctness of the resulting proof script using the Agda proof assistant.

**4.1.4.** Martin-Löf Type Theory (often called Intuitionistic Type Theory) is the formal system at the heart of the Homotopy Type Theory [45] program. Type theory can serve as a self-contained alternative to classical first-order logic and ZFC Set Theory as a foundation for mathematics, and many popular proof assistants and interactive theorem proving tools (such as Agda, Coq, Lean, NuPRL) use Martin-Löf Type Theory or other closely related type theories as their foundation. In particular, the Coq proof assistant played an essential role in the celebrated computer-verified proofs of the Appel-Haken [17] and Feit-Thompson theorems [18].

**4.1.5.** Agda is a pure functional dependently-typed programming language introduced by Ulf Norell [36] and developed chiefly at Chalmers University, Gothenburg. Through a correspondence in the Curry-Howard style (see [15]-Chapter 3 for an overview), one can express mathematical proofs as Agda programs by considering the type of an Agda program as a mathematical statement and a valid program of that type as a mathematical proof of the statement. The type system of Agda contains as a subset all the usual constructions of Martin-Löf Type Theory, so one can formulate proofs in Martin-Löf Type Theory inside the language of Agda, then use Agda's type checker to verify their correctness. Agda works as a *proof assistant*, as opposed to an automated theorem prover: it does not generate proofs by itself, but verifies proof scripts that have been encoded into its programming language by a human mathematician. Agda provides many high-level features including data type definitions, universe polymorphism, implicit arguments, pattern matching, and an interactive environment to facilitate program/proof development in Martin-Löf Type Theory.

**4.1.6.** Agda has been used as a proof assistant in over 200 published works in computer science and in mathematics (see the official Agda website [10] for a full list). This includes formalized results about fundamental groups in Homotopy Type Theory [39] and isomorphism theorems in Universal Algebra [19] among others. Most importantly, Xu [49] developed an Agda formalization of the functional interpretation [46] of nonstandard Heyting arithmetic **H** (a weak subsystem of Nelson's Internal Set Theory). These preceding developments made Agda a salient choice for our own mechanization work.

**4.1.7.** While earlier publications dealing with Internal Set Theory in Agda focused exclusively on results *about* subsystems of Internal Set Theory and used Martin-Löf Type Theory as a meta-theory for their investigations, our current development involves working directly inside an extension of Martin-Löf Type Theory that can faithfully represent and handle proofs that rely on the Idealization, Standardization and Transfer principles of Internal Set Theory. To the best of our knowledge, no other proof in Internal Set Theory has been formalized in such a way to date.

**4.1.8.** Martin-Löf himself proposed non-standard-analytic supplements to his type theory, reminiscent of the methods used in contemporary research on guarded types. However, these extensions do not allow us to transcribe proofs written in Nelson-style Internal Set Theory into type theory. Hence, we propose our own extensions, which augment Martin-Löf Type Theory with a hierarchy of universes for external propositions, along with an external standardness predicate. The direct goal of these extensions is to serve as a foundation for our Agda proof of Theorem 2.3.9.

## Type theory, intuitively

**4.1.9.** In this chapter we present our extensions to Martin-Löf Type Theory and describe how to work with the extensions using the features available in Agda. We strive to keep our presentation mostly self-contained and accessible to those without previous experience with type theories. Due to space constraints, we cannot expect to succeed in this endeavor. The first few chapters of the IAS book on *Homotopy Type Theory* [45] should provide adequate descriptions and further pointers to understand the details we elide in our presentation. Those readers who enjoy category theory may prefer to start with Hofmann's[1] *Syntax and Semantics of Dependent Types* [22] instead. For a full, syntactic introduction to a specific formalization of Martin-Löf Type Theory, we recommend [35]-Section 1.3. Do note however that modern versions of the Agda proof assistant eschew the cumulative[2] hierarchy presented there in favor of universe polymorphism [43]. For the sake of simplicity and readability, we omit all discussion and formalism related to universe polymorphism from our thesis.

**4.1.10.** Martin-Löf Type Theory belongs to the family of *typed $\lambda$-calculi*, so we should begin our overview by discussing the intuitive meaning of the primary operations of the $\lambda$-calculus, abstraction and application (substitution). The terms, judgments rules

[1] Hofmann and Voevodsky, two larger-than-life figures in the field of computer-verified proofs, both died tragically while our work was in progress.

[2] We really do not want a cumulative hierarchy in our work: we could not have $\mathbf{Set}_\omega$ consist of external predicates anymore, as that would contradict our type-theoretic Standardization axiom. To fix the issue, we would have to introduce a disjoint external hierarchy, which essentially doubles the number of rules for the system.

and proof trees given in this subsection serve only as examples to illustrate general principles of $\lambda$-calculi: they *do not* form part of the extended Martin-Löf Type Theory presented in the remainder of the section!

**4.1.11.** When we treat an expression as a function, we must clearly identify one of the variables occurring in the expression as the argument of the function. In mathematics, one usually uses arrow notation for this purpose, writing $x \mapsto x^2$ for the squaring function (unless the function already has a name, e.g. when one writes the sine function as sin instead of $x \mapsto \sin x$). We understand that whatever meaning the variable $x$ had outside the scope of this notation disappears in the expression that follows. For example, even if we had $x = 1$, the expression $x \mapsto x^2$ would not denote $1^2 \in \mathbb{R}$, but some function $\mathbb{R} \to \mathbb{R}$ that squares its argument; similarly, we would not distinguish between the functions $x \mapsto x^2$ and $y \mapsto y^2$. Logically speaking, $x \mapsto \ldots$ *binds* the variable $x$ in $\ldots$, the same way the quantifier binds $x$ in $\forall x \in \mathbb{R}.x^4 > 0$ or the integral sign binds $x$ in the indefinite integral $\int x^5\,dx$. As customary in logic, we refer to “not bound” variables as *free*, and to each expression we associate its set of free variables the obvious way. E.g. when $+$ denotes a constant symbol, then the set of free variables of $x + y$ consists of $x$ and $y$, but the set of free variables of $\lambda y.x + y$ consists of $x$ only. In the $\lambda$-calculus, the $\lambda$ symbol performs the role taken by $\mapsto$ in ordinary mathematical notation, so one would write the squaring function as $(\lambda x.x^2)$.

**4.1.12.** To use a function, one applies it to an argument. This gives rise to the process of application, which the syntax of the $\lambda$-calculus represents by juxtaposition: one writes the application of $f$ to $x$ as $f\ x$. Writing application as juxtaposition optimizes for legibility, but one could equally well have written $\mathrm{app}(f, x)$, or $f(x)$ - as one would do in ordinary mathematics. So one might write $(\lambda x.x^2)\,3$ to denote the application of the squaring function to the number symbol 3.

**4.1.13.** Application and $\lambda$-abstraction form a part of the essential core syntax of every $\lambda$-calculus: the terms of every $\lambda$-calculus include at least these two constructions, along with other ones specific to the calculus under consideration. By convention, we omit superfluous parentheses in terms the following manner:

- $\lambda x.\lambda y.P$ stands for $(\lambda x.(\lambda y.P))$,
- $F\ X\ Y$ stands for $((F\ X)\,Y)$.

**4.1.14.** Distinct from application, one has reduction, the passage from e.g. $(\lambda x.x + x)\,3$

to $3+3$. We define reduction using a substitution operation: given an expression $P$ the reduction of the application $(\lambda x.P)\,t$ gives $P[x \leftarrow t]$, where $t[x \leftarrow t]$ denotes the substitution of the expression $t$ for each occurrence of the variable $x$ in the term $t$ (note that $[-]$ does not belong to the syntax: it counts as an instruction meaning "perform the substitution", not "add $[-]$ to the formula"). We have one complication, so-called variable capture: $(\lambda x.\lambda y.x+y)\,y$ should not reduce to $(\lambda y.y+y)$ but rather to $(\lambda y'.y+y')$. We leave the proper definition of such a *capture-avoiding substitution* operation as an exercise for the reader. In the next subsection, where we give the formal definition of type theory, we represent computational rules such as reduction (and reductions done in reverse) using the notion of definitional equality.

**4.1.15.** Martin-Löf Type Theory is a *typed* $\lambda$-calculus. Typed calculi are deductive systems, in which we deduce *typing judgments* using a collection of permitted *inference rules*. Given two terms of the calculus, $t, T$, a typing judgment has the form $t : T$, which we read as "*the term t has type T*". One sees typing judgments as largely analogous to set membership, for example one might make typing judgments such as $(\lambda x.x^2) : \mathbb{Q} \to \mathbb{Q}_+$ or $\sqrt{2} : \mathbb{R}$. However, unlike set membership, which forms part of the term syntax of set theory, a typing judgment is not itself a term. For example, $\neg(x : \mathbb{R})$ does not count as a judgment, or even a syntactically well-formed expression. Inference rules have the form

$$\frac{H_1 \quad \dots \quad H_n}{C}\ \textsc{Name of rule}$$

where $H_i$ and $C$ denote judgments. The rule above says that once we have made all the judgments $H_i$, we are allowed to make the judgment $C$. As an example, take the inference rule

$$\frac{f : A \to B \qquad x : A}{f\,x : B}\ \textsc{Fun-Elim}$$

which states that for any $A, B$ variable symbols, if we judge $f$ to have type $A \to B$ (a function from $A$ to $B$), and we judge $x$ to have type $A$, then we can judge the term $(f\,x)$ to have type $B$. A *derivation* (or *proof*) of a judgment is a rooted tree constructed using such inference rules, with the *conclusion* of the proof sitting at the root of the tree. E.g.

$$\dfrac{\dfrac{\dfrac{}{\mathrm{add} : \mathbb{R} \to (\mathbb{R} \to \mathbb{R})}\ \textsc{Real-Add} \qquad \dfrac{}{1 : \mathbb{R}}\ \textsc{Real-One}}{\mathrm{add}\,1 : \mathbb{R} \to \mathbb{R}}\ \textsc{Fun-Elim} \qquad \dfrac{}{1 : \mathbb{R}}\ \textsc{Real-One}}{\mathrm{add}\,1\,1 : \mathbb{R}}\ \textsc{Fun-Elim}$$

forms a proof tree of conclusion add 1 1 : $\mathbb{R}$ and uses three different inference rules, REAL-ADD, REAL-ONE and FUN-ELIM. The reader should write down proper formulations of these rules as an exercise; the reader interested in learning[3] even more about proof trees is referred to to Girard's excellent *Proofs and Types* [15].

**4.1.16.** In a typed setting, free variables require special care: barring further information, how does one make any kind of type judgment about the type of $f\ x$ without knowing the type of the variable $x$?! Naively, one might think that annotating each free variable with its type (e.g. writing $x_{:\mathbb{N}}$ instead of $x$) would solve this problem. In practice, this does not work, since the set of valid types depends on the judgments that have been made previously. For example, writing $x_{:y}$ is valid if we already know that $y$ takes one of the values $\mathbb{R}, \mathbb{N}$, but not valid if $y$ might take a value that does not count as a type, say 42. De Bruijn [11] gave a satisfactory solution to this issue in the form of *contextual judgments*. A context $\Gamma$ consists of a list of typed variables such that the free variables of each type appear earlier in the list than the type itself. A contextual typing judgment has the form $\Gamma \vdash t : T$, where all the free variables of $t$ and $T$ appear in the context $\Gamma$. We phrase our formalization of (extended) Martin-Löf type theory purely in terms of contextual judgments.

## 4.2 Extended type theory

**4.2.1.** Here we present an extended variant of Martin-Löf Type Theory that has the same relationship to ordinary Martin-Löf Type Theory as Internal Set Theory has to Zermelo-Fraenkel Set Theory, and can cope with Nelson-style reasoning used in the proof of Theorem 2.3.9. The main innovations of our proposed system include a hierarchy of universes indexed by small ordinals that lets us treat external sets and predicates such as st$(-)$, and a new kind of judgment that allows for a uniform treatment of Transfer schemata. Moreover, we identify a subsystem[4] of our extended type theory that we can effectively encode into Agda, allowing us to computer-verify our proof without having to extend or modify the Agda proof assistant.

[3] According to the Japanese proverb, the best way to learn proof trees is to have learned them ten years ago.

[4] This subsystem is sufficient for representing the proof of Theorem 2.3.9, but in principle the same proof could be carried out in weaker subsystems. In particular, we never invoke Idealization or internal-to-external Transfer principles.

## Contexts, terms and judgments

**4.2.2.** In the following, we assume an inexhaustible (at the very least countable) supply of *variable symbols*, which we usually denote by lower-case letters from the very end of the English alphabet.

**4.2.3. Definition.** We call an ordinal $\lambda$ such that $\lambda < \omega + \omega$ a *universe level* or *level* for short. We usually use the letters $i, j, \ell$ or $m$ for variables that range over universe levels.

**4.2.4. Definition.** In the following, the variables $\Gamma, \Delta$ range over contexts, while $s, t, S$ range over terms. We distinguish four sorts of *judgments* in our type theory:

1. Judgments $\Gamma \vdash$ read as "$\Gamma$ *forms a valid context*".
2. Judgments $\Gamma \vdash t : S$ read as "*the term $t$ has type $S$ in the context* $\Gamma$".
3. Judgments $\Gamma \sim \Delta \vdash$ read as "$\Gamma$ *and* $\Delta$ *denote the same context by definition*".
4. Judgments $\Gamma \vdash s \sim_S t$ read as "*the expressions $s$ and $t$ denote the same inhabitant of type $S$ in the context* $\Gamma$ *by definition*",
5. Judgments $\Gamma \vdash s \Leftrightarrow_\ell t$ read as "*the terms $s$ and $t$ form a transfer pair at universe level $\ell$*".

**4.2.5.** We define the contexts, terms, inference rules and proof trees of type theory in terms of each other[5] via a single, gargantuan, mutually inductive definition. The next few pages contain multiple Definition blocks (from 4.2.8 to 4.2.35), but one should consider them as fragments of a single long definition, which does not conclude until we account for every single rule.

**4.2.6. Definition.** We maintain a distinction between the full calculus and its safe fragment. We say that a derivation *belongs to the safe fragment* if it does not contain any rule whose name contains the symbol $\star$. Some rules require a safety assumption on some of their premises: we mark these assumptions by writing $\vdash^s$ instead of $\vdash$ in the turnstile of the hypothesis. For example, if we write

$$\frac{\Gamma \vdash^s A : \mathbf{Set}_1 \qquad \Gamma \vdash x : A}{x : \mathbf{Set}_0, \Gamma \vdash} \star \text{ example rule (not an actual rule)}$$

[5] Is such a definition circular? No, for the same reason that BNF grammars define sets [28].

then the example rule requires that the derivation of its left premise $\Gamma \vdash A : \mathbf{Set}_1$ occur in the safe fragment (i.e. not use rules marked with the $\star$ symbol), while the derivation of the right premise $\Gamma \vdash x : A$ may use any rule, including those marked with $\star$. Similarly, since the name of the example rule itself contains the $\star$ symbol, any premise of any rule marked with $\vdash^s$ cannot use the example rule in its derivation.

## Rules: Contexts and variables

**4.2.7.** First we describe the rules of context formations. These rules have conclusions labeled with judgments of the form $\Gamma$. Recall that we read these judgments as "$\Gamma$ *forms a valid context*". For the sake of readability, the empty context $\emptyset$ receives special treatment in the syntax: we simply write $x_1 : T_1$ to denote the context $\emptyset, x_1 : T_1$.

**4.2.8. Definition.** Using the variable conventions discussed above and in the preceding definitions, we take the following *context formation rules* in our type theory.

$$\frac{}{\emptyset \vdash}\ \textsc{Ctx-Nul} \qquad \frac{\Gamma \vdash A : \mathbf{Set}_i}{\Gamma, x : A \vdash}\ \textsc{Ctx-Ext}$$

In the rule CTX-EXT, we require that the variable $x$ not occur in the context $\Gamma$. We take the following *variable introduction rule*:

$$\frac{\Gamma, x : A, \Delta \vdash}{\Gamma, x : A, \Delta \vdash x : A}\ \textsc{Var}$$

The presentation of the rules continues in Definition 4.2.10.

## Rules: Context equality

**4.2.9.** The rules presented in this subsection pertain to judgments of the form $\Gamma \sim \Delta$, read as "$\Gamma$ *and* $\Delta$ *denote the same context by definition*". These rules ensure that $\sim$ behaves like an equivalence relation, and tell us when we can consider two contexts equal. Other rules will ensure that we can substitute equal contexts for each other (recall that we give these definitions mutually inductively, as described in Section 4.2.5).

**4.2.10. Definition.** We take the following *context equality rules*.

$$\frac{}{\emptyset \sim \emptyset \vdash}\ \textsc{Ceq-Null} \qquad \frac{\Gamma \sim \Delta \vdash \qquad \Gamma \vdash A \sim B : \mathbf{Set}_i}{\Gamma, x : A \sim \Delta, x : B \vdash}\ \textsc{Ceq-Extn}$$

$$\frac{\Gamma \sim \Delta \vdash}{\Delta \sim \Gamma \vdash}\ \textsc{Ceq-Symm} \qquad \frac{\Gamma \sim \Pi \vdash \qquad \Pi \sim \Delta \vdash}{\Gamma \sim \Delta \vdash}\ \textsc{Ceq-Tran}$$

Mirroring the constraints of the rule CTX-EXT, we require that the variable $x$ not occur in the contexts $\Gamma$ and $\Delta$ within the rule CEQ-EXTN. The presentation of the rules continues in Definition 4.2.12.

## Rules: Term equality

**4.2.11.** Recall that we read judgments of the form $\Gamma \vdash s \sim_S t$ as "*the expressions s and t denote the same term of type $S$ in the context* $\Gamma$ *by definition*". Type theory has its own internal definition of equality between objects (equality types or path types); one must not confuse those with the equality *judgments* presented here, which concern only the equations that hold between terms "by definition". We always allow replacement of definitionally equal terms and contexts with each other in any part of any judgment, while eliding some of the congruence rules asserting this fact in our presentation.

**4.2.12. Definition.** We take the following *term equality rules*.

$$\frac{\Gamma \vdash t : A}{\Gamma \vdash t \sim_A t}\ \textsc{Teq-Refl} \qquad \frac{\Gamma \vdash s \sim_A t}{\Gamma \vdash t \sim_A s}\ \textsc{Teq-Symm}$$

$$\frac{\Gamma \vdash s \sim_A p \qquad \Gamma \vdash p \sim_A t}{\Gamma \vdash t \sim_A s}\ \textsc{Teq-Tran}$$

$$\frac{\Gamma \sim \Delta \qquad \Gamma \vdash A \sim_{\mathbf{Set}_i} B \qquad \Gamma \vdash t : A}{\Delta \vdash t : B}\ \textsc{Teq-SubT}$$

$$\frac{\Gamma \sim \Delta \qquad \Gamma \vdash A \sim_{\mathbf{Set}_i} B \qquad \Gamma \vdash s \sim_A t}{\Delta \vdash s \sim_B t}\ \textsc{Teq-SubE}$$

The presentation of the rules continues in Definition 4.2.17.

## Rules: Universes and Type Formation

**4.2.13.** The type theory of Agda (strictly speaking, Martin-Löf Type Theory with a stratified hierarchy of large types) comes equipped with a hierarchy of universes

$$\mathbf{Set}_0 : \mathbf{Set}_1 : \mathbf{Set}_2 : \ldots$$

Generally, we can think of $\mathbf{Set}_0$ as the "set of all (small) sets", and judgments $S : \mathbf{Set}_0$ as stating "*$S$ is a set*". Under a Curry-Howard interpretation, one might as well read this as "*$S$ is a proposition*". The same way set theories have to avoid constructing the set of all sets, type theory cannot admit $\mathbf{Set}_\lambda : \mathbf{Set}_\lambda$ on pain of contradiction: if we take such a rule, Girard's paradox (a variant of the Burali-Forti paradox) makes the resulting system inconsistent [24]. To avoid contradiction while giving a type to $\mathbf{Set}_0$, we can introduce the universe hierarchy, and take $\mathbf{Set}_0 : \mathbf{Set}_1$, $\mathbf{Set}_1 : \mathbf{Set}_2$ etc.

**4.2.14.** Internal Set Theory requires a strict separation between internal predicates/propositions and external ones, such as the proposition $\mathrm{st}(x)$. Uses of the latter usually fall under much stricter rules (e.g. not available for use within induction arguments, as in 1.1.10). We shall use a second hierarchy of universes, indexed by the levels $\omega, \omega+1, \ldots,$ for these external predicates and propositions.

**4.2.15.** We use ordinal indices for the external hierarchy as a notational convenience, not because of some deep relationship between external predicates and the ordinal hierarchy. Indeed, since our type theory does not have cumulativity, there is *no* type-theoretic relationship between $\mathbf{Set}_0$ and $\mathbf{Set}_\omega$; we could treat internal and external sets as two completely disjoint hierarchies and use the notation $\mathbf{ESet}_0 : \mathbf{ESet}_1 : \ldots$ for the latter. The reasons for not doing this are two-fold. The first is desire for parsimony: the standard ordinal operations max and $+$ make for a shorter presentation, and not having to include separate rules for the **Set**- and **ESet**-hierarchies essentially halves the number of necessary rules. The second consideration is much more pragmatic: we wish to check our proofs using an unmodified Agda proof checker, and current versions of Agda already have an option for doing some "unsafe" things with $\mathbf{Set}_\omega$ without destroying compatibility with standard universe-polymorphic Agda code.

**4.2.16.** Type formation rules control how one can introduce new types. Like universe formation rules, the conclusion of such rules have the form $\Gamma \vdash t : \mathbf{Set}_i$ for some $i$, asserting that the term $t$ inhabits some universe of types. One can regard the universe formation rules themselves as special type formation rules for the types $\mathbf{Set}_i$. We give the type formation rules for each primitive type of the theory in its respective section.

**4.2.17. Definition.** We admit the following *universe formation rules*:

$$\frac{\Gamma \vdash}{\Gamma \vdash \mathbf{Set}_{\ell} : \mathbf{Set}_{\ell+1}} \text{ Univ-Int}$$

$$\frac{\Gamma \vdash}{\Gamma \vdash \mathbf{Set}_{\omega+\ell} : \mathbf{Set}_{\omega+\ell+1}} \star \text{ Univ-Ext}$$

The variable $\ell$ ranges over universe levels satisfying $\ell < \omega$, the $+$ symbol denotes the usual addition operation on the ordinals. Notice that we never have $\mathbf{Set}_i : \mathbf{Set}_\lambda$ for any limit ordinal $\lambda \in \{0, \omega\}$. The presentation of the rules continues in Definition 4.2.19.

## Rules: Dependent function type

**4.2.18.** Here we introduce the dependent function type $\forall x : A.B$. We can get a fairly close set-theoretic analogue of this type in classical Zermelo-Fraenkel Set Theory by taking an $A$-indexed family of sets $B_x$, and forming the set

$$P = \left\{ f : A \to \bigcup_{x \in A} B_x \;\middle|\; \forall x \in A. f(x) \in B_x \right\}.$$

In set theory, we would denote the set $P$ as $\prod_{x \in A} B_x$, and refer to it as the infinite *Cartesian product* of the family $B_x$. In type theory, the dependent function type loosely corresponds to this set $P$. As such, we might denote the dependent function as $(x : A) \to B$ or even as a product $\prod_{x : A} B$; through the Curry-Howard correspondence, we can identify dependent products with (higher-order) universal quantification, and since we take this perspective, we shall write it as $\forall x : A.B$. When the variable $x$ does not occur at all in the term $B$, we write $A \to B$ (like function spaces in Set Theory, or like implication $\to$ via the Curry-Howard correspondence). Agda provides some form of support for all of these notations. The application operation discussed in Section 4.1.12 provides the elimination rule for dependent functions, while $\lambda$-abstraction acts as the introduction rule. The computation rules formalize reduction by substitution.

**4.2.19. Definition.** We admit the following *dependent function rules*:

$$\frac{\Gamma \vdash A : \mathbf{Set}_i \qquad \Gamma, x : A \vdash B : \mathbf{Set}_j}{\Gamma \vdash (\forall x : A.B) : \mathbf{Set}_{\max\{i,j\}}} \ \textsc{Dfun-Form}$$

$$\frac{\Gamma, x : A \vdash t : B}{\Gamma \vdash \lambda x.t : \forall x : A.B} \ \textsc{Dfun-Intr}$$

$$\frac{\Gamma \vdash f : \forall x : A.B \qquad \Gamma \vdash t : A}{\Gamma \vdash f\,t : B[x \leftarrow t]} \ \textsc{Dfun-Elim}$$

$$\frac{\Gamma, x : A \vdash t : B \qquad \Gamma \vdash s : A}{\Gamma \vdash (\lambda x.t)s \sim_{B[x \leftarrow s]} t[x \leftarrow s]} \ \textsc{Dfun-Comp}$$

where the variables $i, j$ range over all possible universe levels, including levels over $\omega$ and $t[x \leftarrow s]$ denotes the capture-avoiding substitution of the term $s$ for each occurrence of the variable $x$ in the term $t$. The presentation of the rules continues in Definition 4.2.21.

## Rules: Dependent sum type

**4.2.20.** Similarly to the set-theoretic analogue of $\forall x : A.B$, we can approximate the meaning of the dependent sum type $\exists x : A.B$ very well in classical ZFC Set Theory by starting with an $A$-indexed family of sets $B_x$, and forming the set

$$P = \left\{ (a, b) \in A \times \bigcup_{x \in A} B_x \,\middle|\, b \in B_a \right\}$$

using Comprehension and Union. For the two-element index set $A = \{1, 2\}$, the construction gives the disjoint union $B_1 \uplus B_2$, and for a constant family $B_x = B$ the binary Cartesian product $A \times B$. Indeed, if $x$ does not occur in $B$, we will write $A \times B$ for $\exists x : A.B$. Through the Curry-Howard correspondence, we can identify dependent sums with (higher-order) existential quantification, and $A \times B$ with the conjunction $A \wedge B$. The introduction rule corresponds to the formation of an ordered pair: if $t : A$ and $s : B_t$, then we have $(t, s) : \exists x : A.B_x$. The elimination rules correspond to coordinate projections.

**4.2.21. Definition.** We admit the following *dependent sum rules*:

$$\frac{\Gamma \vdash A : \mathbf{Set}_i \qquad \Gamma, x : A \vdash B : \mathbf{Set}_j}{\Gamma \vdash (\exists x : A.B) : \mathbf{Set}_{\max\{i,j\}}} \ \textsc{Dsum-Form}$$

$$\frac{\Gamma \vdash t : A \qquad \Gamma \vdash s : B[x \leftarrow t]}{\Gamma \vdash (t,s) : (\exists x : A.B)} \ \textsc{Dsum-Intr}$$

$$\frac{\Gamma \vdash t : (\exists x : A.B)}{\Gamma \vdash \pi_1 t : A} \ \textsc{Dsum-ElimL} \qquad \frac{\Gamma \vdash t : (\exists x : A.B)}{\Gamma \vdash \pi_2 t : B[x \leftarrow \pi_2 t]} \ \textsc{Dsum-ElimR}$$

$$\frac{\Gamma \vdash t : A \qquad \Gamma \vdash s : B[x \leftarrow t]}{\Gamma \vdash \pi_1(t,s) \sim_A t} \ \textsc{Dsum-CompL}$$

$$\frac{\Gamma \vdash t : A \qquad \Gamma \vdash s : B[x \leftarrow t]}{\Gamma \vdash \pi_2(t,s) \sim_{B[x \leftarrow t]} s} \ \textsc{Dsum-CompR}$$

where as usual $t[x \leftarrow s]$ denotes the substitution of the term $s$ for each occurrence of the variable $x$ in the term $t$. The presentation of the rules continues in Definition 4.2.24.

## Rules: Empty type

**4.2.22.** We call $\perp$ the *empty type*. It corresponds (unsurprisingly) to the empty set in set theory. Through the Curry-Howard perspective, $x : \perp$ acts as a proof of a pure contradiction; as such, we can represent negation as $A \to \perp$, and we sometimes abbreviate the latter as $\neg A$. The elimination rule for the empty type corresponds to the principle of explosion: if we manage to produce a proof of a contradiction, *anything* follows. The empty type lacks inhabitants, and so it does not have any introduction rules.

**4.2.23.** Due to the presence of the standardness predicate `st`, and in line with Section 1.1.10, our extended type theory restricts inductive elimination rules to internal levels of the universe hierarchy ($\mathbf{Set}_i$ for $i < \omega$). This limitation does not affect the elimination rule for the empty type, which remains valid for all $i$, including $i \geq \omega$.

**4.2.24. Definition.** We admit the following *empty type rules*:

$$\frac{\Gamma \vdash}{\Gamma \vdash \perp : \mathbf{Set}_0} \ \textsc{Empt-Form}$$

$$\frac{\Gamma \vdash t : \bot \qquad \Gamma \vdash A : \mathbf{Set}_i}{\Gamma \vdash \texttt{absurd}\ A\ t : A}\ \textsc{Empt-Elim}$$

where the variable $i$ ranges over all universe levels. The presentation of the rules continues in Definition 4.2.27.

## Rules: Equality type

**4.2.25.** Definitional equality $\sim$ expresses the computation rules associated with types that hold by definition; Agda will perform such substitutions automatically using simple term rewriting. The *propositional equality types* have a different purpose: they serve as types for equality proofs, internal to the theory. Under a Curry-Howard interpretation, we read $p : a =_S b$ as "$p$ proves that the inhabitant $a$ of type $S$ equals the inhabitant $b$ of the same type". Agda does not perform substitutions along propositional equalities automatically. The introduction rule for the equality type states the reflexivity of equality (for each $x : T$, $x =_T x$).

**4.2.26.** One can choose between multiple different elimination rules for the equality type. We take the strongest elimination rule, Streicher's rule K as our equality elimination rule since it's the default option in Agda's type theory as well. Our formalized proof of Theorem 2.3.9 does not rely on this choice in any form and would work even if we took a weaker elimination rule (such as the elimination rule J commonly used in Homotopy Type Theory). However, we require that the elimination rule permit elimination into any universe level, including levels $i \geq \omega$ of the external hierarchy, since we need the capability of transporting standardness predicate. That is, if we have a proof of $x = y$, and a proof of $\texttt{st}(x)$, we want to have some way to obtain the conclusion $\texttt{st}(y)$. At first glance, this may seem like a departure from Internal Set Theory. As a matter of fact, Internal Set Theory does put forth an identical requirement, but sweeps it under the rug of first-order logic: $\forall x.\forall y.x = y \wedge \mathrm{st}(x) \to \mathrm{st}(y)$ does hold in Internal Set Theory, not as an axiom of Internal Set Theory, but as an axiom of first-order logic. Since type theory acts as its own underlying logic, it has to make provision for this requirement explicitly.

**4.2.27. Definition.** We admit the following *equality type rules*:

$$\frac{\Gamma \vdash A : \mathbf{Set}_i \qquad \Gamma \vdash t : A \qquad \Gamma \vdash s : A}{\Gamma \vdash (t =_A s) : \mathbf{Set}_i}\ \textsc{Eqt-Form}$$

$$\frac{\Gamma \vdash t : A}{\Gamma \vdash \texttt{refl}\, A\, t : (t =_A t)}\ \textsc{Eqt-Intr}$$

$$\frac{[x1] \quad [x2] \quad [x3] \quad [x4] \quad [x5]}{\Gamma \vdash \texttt{K}\ A\ (\lambda x.\lambda p.C)\ t\ q\ P : C[p \leftarrow P, x \leftarrow t]}\ \textsc{Eqt-ElimK}$$

$$\frac{[x1] \quad [x2] \quad [x3] \quad [x4]}{\Gamma \vdash \texttt{K}\ A\ (\lambda x.\lambda p.C)\ t\ q\ (\texttt{refl}\ A\ t) \sim_{C[p \leftarrow \texttt{refl}\ A\ t, x \leftarrow t]} q}\ \textsc{Eqt-Comp}$$

where

x1: $\Gamma \vdash A : \mathbf{Set}_i$

x2: $\Gamma, x : A, p : x =_A x \vdash C : \mathbf{Set}_j$

x3: $\Gamma \vdash t : A$

x4: $\Gamma \vdash q : C[p \leftarrow \texttt{refl}\ A\ t, x \leftarrow t]$

x5: $\Gamma \vdash P : t =_A t$

and the variables $i, j$ range over all universe levels, internal or external. The presentation of the rules continues in Definition 4.2.29.

## Rules: Transfer

**4.2.28.** Since Standardization works over any predicate, internal or external, we can admit it as a proper axiom over all universe levels. Similarly, we can admit Idealization over internal predicates by admitting it as a proper axiom over universe levels below $\omega$. However, the Transfer axioms work only over those internal predicates which have all parameters standard. To capture this purely syntactic restriction, we handle Transfer using a new form of judgment, $\Gamma \vdash A \leftrightarrow_i B$, meaning "one can transfer between $A$ and $B$ of universe level $i$ in the context $\Gamma$". Before we assert the rules governing this new sort of judgment, we have to give meaning to the analogues of the predicate $\mathrm{st}(-)$ of Internal Set Theory, the standardness types `st` $A$ $t$, read as "$t$ is a standard inhabitant of type $A$". This type lives in the external hierarchy, and does not have introduction or elimination rules, as one can use the Transfer axioms directly for all introductions and eliminations of `st`.

**4.2.29. Definition.** In all the rules that follow, the variables $\ell, i, j$ range over universe levels of the internal hierarchy (strictly below $\omega$), and the variable $x$ is fresh with respect to the context $\Gamma$. We admit the following standardness type formation rule.

$$\frac{\Gamma \vdash A : \mathbf{Set}_\ell \qquad \Gamma \vdash t : A}{\Gamma \vdash \mathtt{st}_\ell\ A\ t : \mathbf{Set}_\omega} \star \text{ St-Form}$$

Define the notation $\forall^{st} x : A.B$ as an abbreviation for $\forall x : A.\mathtt{st}\ A\ x \to B$, and similarly $\exists^{st} x : A.B$ as an abbreviation for $\exists x : A.\mathtt{st}\ A\ x \wedge B$. In accordance with the rule St-Form, we need to have $\Gamma \vdash A : \mathbf{Set}_\ell$ for some $\ell < \omega$ before we can write $\forall^{st} x : A.B$. We admit the following *transfer rules*.

$$\frac{\Gamma \vdash^s A : \mathbf{Set}_\ell}{\Gamma \vdash A \Leftrightarrow_\ell A} \star \text{ Trf-Refl}$$

$$\frac{\Gamma \vdash^s A : \mathbf{Set}_\ell \qquad \Gamma, x : A \vdash B \Leftrightarrow_i B'}{\Gamma \vdash (\forall x : A.B) \Leftrightarrow_{\max\{\ell, i\}} (\forall^{st} x : A.B')} \star \text{ Trf-Dfun}$$

$$\frac{\Gamma \vdash A \Leftrightarrow_i A' \qquad \Gamma \vdash B \Leftrightarrow_j B'}{\Gamma \vdash (A \to B) \Leftrightarrow_{\max\{i, j\}} (A' \to B')} \star \text{ Trf-Fun}$$

$$\frac{\Gamma \vdash^s A : \mathbf{Set}_\ell \qquad \Gamma, x : A \vdash B \Leftrightarrow_i B'}{\Gamma \vdash (\exists x : A.B) \Leftrightarrow_{\max\{\ell, i\}} (\exists^{st} x : A.B')} \star \text{ Trf-Dsum}$$

$$\frac{\Gamma \vdash A \Leftrightarrow_i A' \qquad \Gamma \vdash B \Leftrightarrow_j B'}{\Gamma \vdash (A \times B) \Leftrightarrow_{\max\{i, j\}} (A' \times B')} \star \text{ Trf-Sum}$$

The presentation of the rules continues in Definition 4.2.32.

## Rules: Naturals and Finite Lists

**4.2.30.** We call a rule a *proper axiom* if it has the form

$$\frac{\Gamma \vdash}{\Gamma \vdash t : A} \text{ Ax-T}$$

for fixed terms $t, A$ and an arbitrary context $\Gamma$. One can treat introduction and elimination rules for the basic types of the theory ($\mathbb{N}$, `List`, etc) as either proper axioms or pure inference rules; compare e.g.

$$\frac{\Gamma \vdash}{\Gamma \vdash \mathtt{suc} : \mathbb{N} \to \mathbb{N}} \text{ axiom} \qquad \frac{\Gamma \vdash n : \mathbb{N}}{\Gamma \vdash \mathtt{suc}\ n : \mathbb{N}} \text{ inf. rule}$$

Presenting the basic types using pure inference rules results in a more *modular* theory (the definition of $\mathbb{N}$ does not depend on how we define universes and dependent functions, or whether we have them at all), at the cost of longer and more complicated proofs in the meta-theory, and a less clear correspondence with the Agda code. On that account, we opt to introduce the basic types of our type theory as proper axioms (along with formation and computation rules). To save space, we give all the axioms on single lines, e.g. we write the axiom introducing `suc` simply as $\mathtt{suc} : \mathbb{N} \to \mathbb{N}$. Later on, we present the Idealization and Standardization principles the same way as well.

**4.2.31.** The type of natural numbers has two introduction rules, stating that zero is a natural number and the successor of every natural number is also a natural number. The principle of induction gives the elimination rule. We define the type `List` $A$ of finite lists over the type $A$ in a similar fashion, with the elimination rule given by structural induction. In accordance with Section 1.1.10, we have to restrict these induction principles to the universe levels of the internal hierarchy.

**4.2.32. Definition.** Let the variable $\ell$ range over universe levels below $\omega$, and let $i$ range over all universe levels. We admit the following rules for the type of natural numbers (using the notation of Section 4.2.30):

$$\mathbb{N} : \mathbf{Set}_0$$

$$\mathtt{zero} : \mathbb{N}$$

$$\mathtt{suc} : \mathbb{N} \to \mathbb{N}$$

$$\begin{aligned} &\mathtt{induction}_\ell : \forall \varphi : \mathbb{N} \to \mathbf{Set}_\ell . \\ &\qquad \varphi\ \mathtt{zero} \to \\ &\qquad (\forall k : \mathbb{N}. \varphi\ k \to \varphi\ (\mathtt{suc}\ k)) \to \\ &\qquad \forall n : \mathbb{N}. \varphi\ n \end{aligned}$$

along with the computation rules

$$\mathtt{induction}_\ell\ \varphi\ z\ s\ \mathtt{zero} \sim z \text{ and}$$
$$\mathtt{induction}_\ell\ \varphi\ z\ s\ (\mathtt{suc}\ n) \sim s\ (\mathtt{induction}_\ell\ \varphi\ z\ s\ n).$$

We admit the following rules for the type of finite lists.

$$\begin{aligned}
&\mathtt{List}_i : \mathbf{Set}_i \to \mathbf{Set}_i\\
&\mathtt{empty}_i : \forall A : \mathbf{Set}_i.\mathtt{List}_i\ A\\
&\mathtt{cons}_i : \forall A : \mathbf{Set}_i.A \to \mathtt{List}_i\ A \to \mathtt{List}_i\ A\\
&\mathtt{listelim}_{i,\ell} : \forall A : \mathbf{Set}_i.\forall \varphi : \mathtt{List}_i\ A \to \mathbf{Set}_\ell.\\
&\qquad \varphi\ (\mathtt{empty}_i\ A) \to\\
&\qquad \big(\forall a : A.\forall k : \mathtt{List}_i\ A.\varphi\ k \to \varphi\ (\mathtt{cons}_i\ A\ a\ k)\big) \to\\
&\qquad \forall n : \mathtt{List}_i\ A.\varphi\ n
\end{aligned}$$

along with the computation rules

- $\mathtt{listelim}_{i,\ell}\ A\ \varphi\ e\ c\ (\mathtt{empty}_i\ A) \sim e$ and
- $\mathtt{listelim}_{i,\ell}\ A\ \varphi\ e\ c\ (\mathtt{cons}_i\ A\ h\ t) \sim c\ h\ (\mathtt{listelim}_{i,\ell}\ A\ \varphi\ e\ c\ t)$.

We define the list membership predicate $\mathtt{elem}_i : \forall A : \mathbf{Set}_0.A \to \mathtt{List}_i\ A \to \mathbf{Set}_0$ as an abbreviation for the term

$$\lambda A : \mathbf{Set}_i.\lambda e : A.\mathtt{listelim}_{i,\ell}\ A\ (\lambda x.\mathbf{Set}_0)\ \bot\ (\lambda h.\lambda t.\lambda P.\neg(e = h) \to P)$$

When one can deduce the universe level $i$ and the type $A$ from the surrounding text, we often leave these implicit and denote $\mathtt{elem}_i\ A\ x\ n$ as $x \in n$. The presentation of the rules continues in Definition 4.2.35.

**4.2.33. Exercise.** Prove that the list membership predicate $\mathtt{elem}_i$ defined in Definition 4.2.32 satisfies the definitional equalities

$$\mathtt{elem}_i\ A\ e\ (\mathtt{empty}_i\ A) \sim \bot$$
$$\mathtt{elem}_i\ A\ e\ (\mathtt{cons}_i\ A\ h\ t) \sim \neg(e = h) \to \mathtt{elem}_i\ A\ e\ t.$$

Convince yourself of the correctness of the definition (hint: the equalities above state that no $e$ belong to the empty list, and if $e$ belongs to a list starting with the element $a$, then either $a = e$ or $e$ belongs to the tail of the same list).

## Rules: IST Axioms

**4.2.34.** We now give the Idealization, Standardization and Transfer axioms. Thanks to the extended hierarchy of universes, an external predicate on the type $A$ corresponds simply to a function of signature $A \to \mathbf{Set}_\omega$, while functions of signature $A \to \mathbf{Set}_0$ always give internal predicates. This allows us to admit both Idealization and Standardization as proper axioms without any complication. Unfortunately, the same trick would not work for Transfer axioms, since they require not only the internality of their predicates, but also that said predicates do not contain any non-standard parameters (otherwise we could transfer the true sentence $\forall^{st} n : \mathbb{N}.n < \omega$ and conclude $\omega < \omega$). Consequently, we need to use the transfer judgments introduced in Definition 4.2.29 to define the valid instances of Transfer axioms.

**4.2.35. Definition.** Let the variables $\ell, m, k, i, j$ range over universe levels of the internal hierarchy (strictly below $\omega$). We admit the following *Idealization/Standardization rules* into our type theory (using the notation of Section 4.2.30):

$\star$ $\mathtt{IdealizationF}_{\ell,m,k} : \forall A : \mathbf{Set}_\ell.\forall B : \mathbf{Set}_m.\forall \varphi : A \to B \to \mathbf{Set}_k.$
$(\forall^{st} t : \mathit{List}\ A.\exists b : B.\forall a : A.a \in t \to \varphi\ a\ b) \to$
$\exists b : B.\forall^{st} a : A.\varphi\ a\ b$

$\star$ $\mathtt{IdealizationB}_{\ell,m,k} : \forall A : \mathbf{Set}_\ell.\forall B : \mathbf{Set}_m.\forall \varphi : A \to B \to \mathbf{Set}_k.$
$(\exists b : B.\forall^{st} a : A.\varphi\ a\ b) \to$
$\forall^{st} t : \mathit{List}\ A.\exists b : B.\forall a : A.a \in t \to \varphi\ a\ b$

$\star$ $\mathtt{Standardization}_\ell : \forall A : \mathbf{Set}_\ell.\forall \varphi : A \to \mathbf{Set}_\omega.\exists^{st} \psi : A \to \mathbf{Set}_\ell.$
$\forall^{st} a : A.(\psi\ a \to \varphi\ a) \wedge (\varphi\ a \to \psi\ a)$

We take the following *Transfer axioms*.

$$\frac{\Gamma \vdash A \Leftrightarrow_j A' \qquad \Delta \vdash}{\Delta \vdash \mathtt{TraL}_{\Gamma,A,A'} : \forall^{st}[\Gamma].(A \to A')} \star\ \text{AX-TRAL}$$

$$\frac{\Gamma \vdash A \Leftrightarrow_j A' \qquad \Delta \vdash}{\Delta \vdash \mathtt{TraR}_{\Gamma,A,A'} : \forall^{st}[\Gamma].(A' \to A)} \star\ \text{AX-TRAR}$$

where the variable $x$ is fresh with respect to the context $\Gamma$, and $\forall^{st}[\Gamma].t$ is defined via the following structurally recursive clauses:

- $\forall^{st}[\emptyset].t$ denotes $t$;
- $\forall^{st}[\Gamma, x : A].t$ denotes $\forall^{st}[\Gamma].(\forall^{st} x : A.t)$.

This concludes the presentation of the rules of our extended type theory.

## 4.3 Syntactic properties

**4.3.1.** Calling a formal system a "type theory" conjures up mental images of certain desirable syntactic properties that such theories tend to satisfy. These include type-theory-specific features such as the substitution property and the existence of canonical forms, as well as more general desiderata such as monotonicity and consistency. Here we discuss which of these properties our newly defined type theory enjoys.

**4.3.2.** We call a term of type $\mathbb{N}$ a canonical natural number if we can write it purely in terms of `zero` and `suc`. Recall that a theory has canonical forms if we can computationally turn every derivation tree with conclusion $\vdash t : \mathbb{N}$ into a derivation tree with conclusion $\vdash t \sim_{\mathbb{N}} n$ for some canonical natural number $n$. Since we intend to use our extended type theory for classical (as opposed to constructive) reasoning, the existence of canonical forms loses its relevance: adding axioms without computation rules (such as the principle of excluded middle or Voevodsky's univalence axiom) destroy canonicity. Indeed, one *does not* have canonical forms in our extended type theory, since the IST axioms lack associated computation rules. More generally, a consistent theory *cannot* have canonical forms in the presence of `IdealizationF`, since one can use the axiom to show the existence of a term $t$ with $\neg\texttt{st}\ \mathbb{N}\ t$, while our type theory proves $\texttt{st}\ \mathbb{N}\ n$ for any canonical natural number $n$ (exercise!).

**4.3.3. Definition.** We say that a type theory enjoys the *monotonicity property* if, given a derivation tree with conclusion $\Gamma \vdash a : A$ and a derivation tree with conclusion $\Gamma, \Delta \vdash$, we can find a derivation tree of $\Gamma, \Delta \vdash a : A$.

**4.3.4. Definition.** We say that a type theory enjoys the *substitution property* if, given a derivation tree with conclusion $\Gamma \vdash a : A$ and a derivation tree with conclusion $\Gamma, x :$

$A, \Delta \vdash t : T$ ($\Gamma, x : A, \Delta \vdash$), we can find a derivation tree with conclusion $\Gamma, \Delta[x \leftarrow a] \vdash t[x \leftarrow a] : T[x \leftarrow a]$ (resp. $\Gamma, \Delta[x \leftarrow a] \vdash$).

**4.3.5. Theorem.** The safe fragment (Definition 4.2.6) of our calculus enjoys the substitution property, the monotonicity property and the following *presupposition properties*:

1. If $\Gamma \vdash^s A : \mathbf{Set}_i$ [has a derivation tree], then [so does] $\Gamma \vdash^s$.

2. If $\Gamma \vdash^s t : A$, then $\Gamma \vdash^s A : \mathbf{Set}_i$ for some level $i < \omega$.

3. If $\Gamma \vdash^s t \sim_A s$, then $\Gamma \vdash^s t : A$ and $\Gamma \vdash^s s : A$.

**Proof.** These properties follow by induction on the length of the derivation tree, combined with a case analysis on the last used rule (see [14]-Lemma 2.1.10 for an example of a proof in this vein).
**Qed.**

**4.3.6. Lemma.** If $\Gamma \vdash t \Leftrightarrow_i v$ has a derivation tree in the full calculus, then $\Gamma \vdash^s t : \mathbf{Set}_i$ has a derivation tree in the safe fragment.

**Proof.** By induction on the derivation tree. If we used the rule $\star$ TRF-REFL as the last rule of our derivation tree, then the tree has the form

$$\dfrac{\overset{\vdots_1}{\Gamma \vdash^s t : \mathbf{Set}_\ell}}{\Gamma \vdash t \Leftrightarrow_\ell t} \star \text{ TRF-REFL}$$

for some $i = \ell < \omega$. Consequently, $\Gamma \vdash^s t : \mathbf{Set}_\ell$ has a derivation tree $\vdots_1$ in the safe fragment.
Otherwise, we must have used one of the following rules: $\star$ TRF-DFUN, $\star$ TRF-FUN, $\star$ TRF-DSUM or $\star$ TRF-SUM. Without loss of generality we consider only the first two of these. In the first case, our tree has the shape

$$\dfrac{\overset{\vdots_1}{\Gamma \vdash^s A : \mathbf{Set}_\ell} \qquad \overset{\vdots_2}{\Gamma, x : A \vdash B \Leftrightarrow_m B'}}{\Gamma \vdash (\forall x : A.B) \Leftrightarrow_{\max\{\ell,m\}} (\forall^{st} x : A.B')} \star \text{ TRF-DFUN}$$

where $t$ has the form $\forall x : A.B$, $v$ has the form $\forall^{st} x : A.B'$ and $i = \max\{\ell, m\}$. Applying the induction hypothesis to the subtree $\vdots_2$, we get a derivation tree $\vdots_{2'}$ of conclusion $\Gamma, x : A \vdash^s B : \mathbf{Set}_m$. Thus, we can construct a proof tree with the desired conclusion $\Gamma \vdash^s \forall x : A.B : \mathbf{Set}_{\max\{\ell,m\}}$ in the safe fragment:

$$\dfrac{\Gamma \vdash^s \overset{\vdots_1}{A} : \mathbf{Set}_\ell \qquad \Gamma, x : A \overset{\vdots_{2'}}{\vdash^s} B : \mathbf{Set}_m}{\Gamma \vdash \forall x : A.B : \mathbf{Set}_{\max\{\ell,m\}}} \ \textsc{Dfun-Form}$$

In the second case, our tree has shape

$$\dfrac{\Gamma \vdash \overset{\vdots_1}{A} \Leftrightarrow_\ell A' \qquad \Gamma \vdash \overset{\vdots_2}{B} \Leftrightarrow_m B'}{\Gamma \vdash (A \to B) \Leftrightarrow_{\max\{\ell,m\}} (A' \to B')} \ \star\ \textsc{Trf-Fun}$$

where $t$ has the form $A \to B$, $v$ has the form $A' \to B'$ and $i = \max\{\ell, m\}$. Applying the induction hypothesis to the subtrees $\vdots_1$ and $\vdots_2$, we get derivation trees $\vdots_{1'}$ and $\vdots_{2'}$ with respective conclusions $\Gamma \vdash^s A : \mathbf{Set}_\ell$ and $\Gamma \vdash^s B : \mathbf{Set}_m$. We can pick a variable $x$ fresh with respect to both contexts $\Gamma$ and $\Delta$, and using the rule Ctx-Ext on the subtree $\vdots_{1'}$ we can obtain $\Gamma, x : A \vdash^s$. Now, by the monotonicity property of the safe fragment (Theorem 4.3.5), we have a derivation tree $\vdots_{2''}$ with conclusion $\Gamma, x : A \vdash^s B : \mathbf{Set}_m$, so we can conclude

$$\dfrac{\Gamma \vdash^s \overset{\vdots_{1'}}{A} : \mathbf{Set}_\ell \qquad \Gamma, x : A \overset{\vdots_{2''}}{\vdash^s} B : \mathbf{Set}_m}{\Gamma \vdash A \to B : \mathbf{Set}_{\max\{\ell,m\}}} \ \textsc{Dfun-Form}$$

which proves our claim.
**Qed.**

**4.3.7. Corollary.** The operation $\forall^{st}[\Gamma]$ introduced in Definition 4.2.35 is well-typed.

**Proof.** We need to verify that if $\Gamma \vdash t \Leftrightarrow_i s$ then for every $x : A$ in $\Gamma$ we have $A : \mathbf{Set}_\ell$ for some $\ell < \omega$. By Lemma 4.3.6 we have $\Gamma \vdash^s t : \mathbf{Set}_i$, so by the presupposition property for the safe fragment (Theorem 4.3.5) we have $\Gamma \vdash^s$. But all derivations in the safe fragment clearly have the desired property.
**Qed.**

**4.3.8. Theorem.** Our extended type theory enjoys the *substitution property*: given a derivation tree with conclusion $\Gamma \vdash a : A$ and a derivation tree with conclusion $\Gamma, x : A, \Delta \vdash t : T$ ($\Gamma, x : A, \Delta \vdash$), we can find a derivation tree with conclusion $\Gamma, \Delta[x \leftarrow a] \vdash t[x \leftarrow a] : T[x \leftarrow a]$ (resp. $\Gamma, \Delta[x \leftarrow a] \vdash$).

**Proof.** The proof by induction proceeds analogously to that of Theorem 4.3.5. Extending a system with proper axioms in the style of Section 4.2.30 cannot break the substitution property, so it suffices to consider only the case where the derivation of

$\Gamma \vdash a : A$ starts with one of the rules $\star$ AX-TRAL or $\star$ AX-TRAR, without loss of generality the former. So the tree has the form

$$\dfrac{\overset{\vdots_1}{\Pi \vdash M \Leftrightarrow_j M'} \qquad \overset{\vdots_2}{\Gamma \vdash}}{\Gamma \vdash \mathtt{TraL}_{\Pi,M,M'} : \forall^{st}[\Pi].(M \to M')} \ \star\ \text{AX-TRAL}$$

If the other derivation tree has the conclusion $\Gamma, x : A, \Delta \vdash$, we distinguish two cases.

- $\Delta$ has the form $\emptyset$. Then we have to produce a derivation tree with conclusion $\Gamma \vdash$; but we already have that, in the form of $\vdots_2$.
- $\Delta$ has the form $\Delta', q : Q$. Then we have to produce a derivation tree with conclusion $\Gamma, \Delta'[x \leftarrow a], q : Q[x \leftarrow a] \vdash$, and we must have had CTX-EXT as the last rule in the derivation of $\Gamma, x : A, \Delta', q : Q \vdash t : T$. This means we have a derivation tree for $\Gamma, x : A, \Delta' \vdash Q : \mathbf{Set}_i$ for some $i$. Applying the induction hypothesis to this derivation yields $\Gamma, \Delta'[x \leftarrow a] \vdash Q[x \leftarrow a] : \mathbf{Set}_i$, so we can finish the proof by using CTX-EXT again.

Now, if the other derivation tree has the conclusion $\Gamma, x : A, \Delta \vdash t : T$, and the last rule of the tree contains a formation, introduction or elimination rule, then the proof proceeds uniformly by applying the induction hypothesis to each premise, then applying the rule again to all the results. Otherwise, the VAR rule occurs as the last rule of the tree. If the variable $t : T$ occurs in either $\Gamma$ or $\Delta$, we can proceed by induction using the previous strategy. Otherwise, the derivation tree has the following form:

$$\dfrac{\overset{\vdots_3}{\Gamma, x : \forall^{st}[\Pi].(M \to M'), \Delta \vdash}}{\Gamma, x : \forall^{st}[\Pi].(M \to M'), \Delta \vdash x : \forall^{st}[\Pi].(M \to M')} \ \text{VAR}$$

and we have to produce a derivation tree with conclusion $\Gamma, \Delta[x \leftarrow \mathtt{TraL}_{\Pi,M,M'}] \vdash \mathtt{TraL}_{\Pi,M,M'} : \forall^{st}[\Pi].(M \to M')$. Using the induction hypothesis on $\vdots_3$, we get a derivation tree $\vdots_{3'}$ with conclusion $\Gamma, \Delta[x \leftarrow \mathtt{TraL}_{\Pi,M,M'}] \vdash$. The derivation tree

$$\dfrac{\overset{\vdots_1}{\Pi \vdash M \Leftrightarrow_j M'} \qquad \overset{\vdots_{3'}}{\Gamma, \Delta[x \leftarrow \mathtt{TraL}_{\Pi,M,M'}] \vdash}}{\Gamma, \Delta[x \leftarrow \mathtt{TraL}_{\Pi,M,M'}] \vdash \mathtt{TraL}_{\Pi,M,M'} : \forall^{st}[\Pi].(M \to M')} \ \star\ \text{AX-TRAL}$$

suffices and concludes our proof.
**Qed.**

**4.3.9. Exercise.** Prove the monotonicity property.

**4.3.10. Proposition.** No consistent extension of our type theory enjoys canonicity.

**Proof.** First we show that for every canonical natural number $n$ our type theory proves $\mathtt{st}_0\ \mathbb{N}\ n$. For canonical $n$ we can easily find a derivation tree $\vdash^s n : \mathbb{N}$. We give the derivation tree for a closed term of type $\exists x : \mathbb{N}.\mathtt{st}_0\ \mathbb{N}\ x \times x =_{\mathbb{N}} n$ on Figure 4.1. Using this term, a transport argument immediately gives a derivation tree for $\vdash \mathtt{st}_0\ \mathbb{N}\ n$.
Find a closed term $f : \forall S : \mathtt{List}\ \mathbb{N}.\exists x : \mathbb{N}.\forall y : \mathbb{N}.y \in S \to \neg(y = x)$ in our type theory (clearly we can already do this in Martin-Löf Type Theory without the extensions). Then the closed term

$$\mathtt{IdealizationF}\ \mathbb{N}\ \mathbb{N}\ (\lambda x.\lambda y.\neg(x = y))\ f$$

has type $\exists x : \mathbb{N}.\forall^{st} y : \mathbb{N}.\neg(x = y)$. Denoting the term by $f'$, we have derivation trees for

$$\vdash \pi_1 f' : \mathbb{N} \text{ and}$$
$$\vdash \pi_2 f' : \forall^{st} y : \mathbb{N}.\neg(\pi_1 f' = y).$$

Using these, we can construct the derivation tree of Figure 4.2 which witnesses the non-standardness of $\pi_2 f'$.
Now, if we had an extension of our type theory that has $\pi_2 f' \sim n$ for some canonical natural number $n : \mathbb{N}$, then we would have proofs of both $\mathtt{st}_0\ \mathbb{N} n$ and $\mathtt{st}_0\ \mathbb{N} \to \bot$, showing the inconsistency of the extension. Therefore, consistent extensions of our type theory do not enjoy canonicity.
**Qed.**

**4.3.11. Proposition.** One can conservatively extend our type theory with the rule

$$\frac{\vdash^s A : \mathbf{Set}_i \qquad \vdash^s t : A}{\Gamma \vdash \mathtt{stcon}_i\ A\ t : \mathtt{st}_i\ A\ t}\ \textsc{St-Con}$$

along with a rule St-Fun realizing the analogue of Lemma 1.1.33.

**Proof.** For the conservativity of St-Con, just notice that the proof of Figure 4.1 does not use any specific fact about $\mathbb{N}$ or about the canonicity of the numeral $n$.

We prove the conservativity of ST-FUN by explicitly constructing a term `stfun` of the required type. Start by picking (exercise!) a derivation tree with conclusion

$$A : \mathbf{Set}_i, B : A \to \mathbf{Set}_j, f : \forall x : A.B, a : A \vdash^s$$
$$(f\ a, \mathtt{refl}\ (B\ a)\ (f\ a)) : \exists y : B\ a.y =_{B\ a} f\ x.$$

Denote the context by $\Gamma$, the term $(f\ a, \mathtt{refl}\ (B\ a)\ (f\ a))$ by $p$, its type by $P$. We can construct the derivation tree

$$\dfrac{\dfrac{\dfrac{\begin{matrix}\vdots \\ \Gamma, y : B\ a \vdash B\ a : \mathbf{Set}_j\end{matrix} \quad \begin{matrix}\vdots \\ \Gamma, y : B\ a \vdash y : B\ a\end{matrix} \quad \begin{matrix}\vdots \\ \Gamma, y : B\ a \vdash f\ x : B\ a\end{matrix}}{\Gamma, y : B\ a \vdash y =_{B\ a} f\ x}\ \text{EQT-FORM}}{\Gamma, y : B\ a \vdash (y =_{B\ a} f\ x) \Leftrightarrow_{\max\{i,j\}} (y =_{B\ a} f\ x)}\ \star\ \text{TRF-REFL}}{\Gamma \vdash (\exists y : B\ a.y =_{B\ a} f\ x) \Leftrightarrow_{\max\{i,j\}} (\exists^{st} y : B\ a.y =_{B\ a} f\ x)}\ \star\ \text{TRF-DSUM}$$

and using the rule $\star$ AX-TRAL we get a closed term

$$t : \forall^{st}[\Gamma].(\exists y : B\ a.y =_{B\ a} f\ x \to \exists^{st} y : B\ a.y =_{B\ a} f\ x).$$

Using this term, we can easily construct a term of type $\forall^{st}[\Gamma].\exists^{st} y : B\ a.y =_{B\ a} f\ x$, and from there on `stfun` of type $\forall^{st}[\Gamma].\mathtt{st}_j\ (B\ a)\ (f\ x)$.
**Qed.**

## Consistency

**4.3.12.** The implementation of Agda assumes that the underlying type theory obeys the substitution property (Theorem 4.3.8) and monotonicity; if these did not hold for our extended type theory, we could not check the proofs using Agda. Proposition 4.3.11 also plays a significant role in the mechanization, by significantly shortening common standardness proofs. At this point, we have all the syntactic properties required to proceed with the mechanization. Before we move on, we take a brief look at consistency and conservative extension results for our proposed calculus.

**4.3.13. Theorem.** Our type theory does not constitute a *conservative* extension of ordinary Martin-Löf Type Theory.

**Proof.** We employ a strategy from Sanders [42] to prove Markov's principle for an arbitrary predicate. By a celebrated result of Coquand and Mannaa [9], ordinary Martin-Löf Type Theory does not prove Markov's principle, so this suffices to prove the non-conservativity of our extension. The argument takes place (informally) inside our extended type theory. Let $A \uplus B$ denote a constructive disjunction operation, say

$$\exists n : \mathbb{N}.(n =_{\mathbb{N}} 0 \to A) \times ((n =_{\mathbb{N}} 0 \to \bot) \to B).$$

Take any standard predicate $P : \mathbb{N} \to \mathbf{Set}_0$, and assume that $\forall n : \mathbb{N}.P\ n \uplus \neg(P\ n)$ and $\neg\forall n : \mathbb{N}.P_n$ hold. Take a nonstandard $\omega : \mathbb{N}$. Assuming $\forall n : \mathbb{N}.n < \omega \to P\ n$ would immediately imply $\forall^{st} n : \mathbb{N}.P\ n$, which would contradict $\neg\forall n : \mathbb{N}.P_n$ after a use of Transfer. Hence, we have $\neg\forall n : \mathbb{N}.n < \omega \to P\ n$. But type theory does prove

$$\forall k : \mathbb{N}.\neg(\forall n : \mathbb{N}.n < k \to P\ n) \to \exists n : \mathbb{N}.\neg(P\ n)$$

and substituting $k = \omega$ immediately gives us

$$\neg(\forall n : \mathbb{N}.n < \omega \to P\ n) \to \exists n : \mathbb{N}.\neg(P\ n).$$

We have established $\neg\forall n : \mathbb{N}.n < \omega \to P\ n$ earlier, so we can conclude $\exists n : \mathbb{N}.\neg(P\ n)$. Since we started with an arbitrary standard predicate $P : \mathbb{N} \to \mathbf{Set}_0$, we have shown

$$\forall^{st} P : \mathbb{N} \to \mathbf{Set}_0.(\forall n : \mathbb{N}.P\ n \uplus \neg(P\ n)) \to \neg(\forall n : \mathbb{N}.P\ n) \to \exists n : \mathbb{N}.\neg(P\ n).$$

Using a quick Transfer argument we get the same conclusion for an arbitrary predicate, which proves Markov's principle.
**Qed.**

**4.3.14.** Given its similarity to Internal Set Theory and our extended type theory, it would be very surprising if our extended type theory would turn out to be inconsistent. That said, a full proof of consistency for our extended type theory seems out of our reach, at least in the near future. While we suspect that (in principle) a proof translation argument done in the style of Nelson [33] (see Proposition 1.1.43) and targeting a carefully chosen classical extension of Martin-Löf Type Theory, will suffice to establish the consistency of our extensions, such an argument presents many technical difficulties. First of all, even classical extensions of type theory lack prenex forms (if $x$ occurs in $C$ then one cannot rewrite $\forall x : (\forall y : A.B).C$ as $\exists y : A.\forall x : B.C$ since the types no longer match). This complicates the formulation of any possible analogue of the Galactic Halo theorem. Even if one finds a way around this particular barrier, one has to face the fact that type theory has many more rules than the first-order logic underlying Zermelo-Fraenkel Set Theory, which makes a Nelson-style proof translation far less convenient. However, such a translation would have a major advantage over the one for Zermelo-Fraenkel Set Theory: while in ZF, the Galactic Halo theorem requires a full Choice principle to realize the quantifier switches, Martin-Löf Type Theory *proves* all these instances of Choice, so the quantifier switch turns out to be innocent, and all the non-constructive content of the translation is concentrated in the Ultrafilter Lemma.

**4.3.15. Proposition.** If we remove the axioms `IdealizationF` and `IdealizationB` from our extended type theory, we obtain a consistent extension of ordinary Martin-Löf Type Theory.

**Proof.** We sketch a proof that our theory with these two axioms removed conservatively extends ordinary Martin-Löf Type Theory extended with the Law of Excluded Middle $LEM_i : \forall A : \mathbf{Set}_i.\neg A \uplus A$. Consider the proof translation that transcribes proofs in the extended theory into proofs of ordinary Martin-Löf Type Theory by sending $\mathbf{Set}_{\omega+\ell}$ to $\mathbf{Set}_\ell$ and $\mathtt{st}\ A\ t : \mathbf{Set}_\omega$ to $\mathtt{zero} =_{\mathbb{N}} \mathtt{zero} : \mathbf{Set}_0$. The interpretation of the Transfer rules and axioms becomes trivial. All we have to do is give an interpretation to the transcribed Standardization axioms

$$\star\,\mathtt{Standardization}_\ell : \forall A : \mathbf{Set}_\ell.\forall \varphi : A \to \mathbf{Set}_0.\exists \psi : A \to \mathbf{Set}_\ell.$$
$$\forall a : A.(\psi\ a \to \varphi\ a) \times (\varphi\ a \to \psi\ a)$$

We can realize this using excluded middle for $\ell > 0$ by considering the following term:

$$\lambda A.\lambda \varphi.\lambda a.$$
$$\mathtt{induction}_\ell(\lambda p.\mathbf{Set}_{\ell-1})(\mathbf{Set}_{\ell-1} \to \bot)(\lambda k.\lambda p.\mathbf{Set}_{\ell-1})(LEM_0(\varphi\ a)) :$$
$$\forall A : \mathbf{Set}_\ell.(A \to \mathbf{Set}_0) \to A \to \mathbf{Set}_0$$

Denote this term $f$. Intuitively, $f$ performs a case analysis on the value of $LEM_i\ (\varphi\ a)$. If it finds $\neg A$, then it returns the uninhabited type $\mathbf{Set}_{\ell-1} \to \bot$, otherwise it returns the inhabited type $\mathbf{Set}_{\ell-1}$. Taking $\psi$ as $f\ A\ \varphi$ allows us to interpret the `Standardization` axioms: the implications hold since $\psi\ a$ has an inhabitant precisely if $\varphi\ a$ does.
**Qed.**

## 4.4 Agda proof

**4.4.1.** We discuss remaining matters related to the formalized proof of Theorem 2.3.9 in this final section. We do not give a syntax reference for the language of Agda: the interested reader can refer to the original article introducting Agda [36], and to the wide range of tutorials available on the official Agda website[6].

### Extended type theory in Agda

**4.4.2.** Agda is a general-purpose proof assistant implementing Martin-Löf Type Theory. It does not know about, and does not incorporate, the extensions we proposed in the previous sections. Fortunately, newer versions of Agda (2.6.0 and above) have features and facilities that we can use to simulate working in our extended type theory, while maintaining fairly wide correctness guarantees.

**4.4.3.** First we have to deal with the question of the extended universe hierarchy introduced in Definition 4.2.17. Normally, Agda supports only finite levels in its universe hierarchy, and does not allow declaring new universes (*sorts*) without modifying the source code of the type checker. However, since version 2.6.0, Agda provides an option `-omega-in-omega` that permits us to treat $\mathtt{Set}\omega$ as a new universe level obeying $\mathtt{Set}\omega : \mathtt{Set}\omega$. This allows us to simulate an external hierarchy by defining $\mathbf{Set}_{\omega+\ell} = \mathbf{Set}_{\omega} = \mathtt{Set}\omega$ for each level $\ell < \omega$. Major caveat: the onus of responsibility of ensuring that one can assign consistent universe levels to all occurrences on the symbol $\mathtt{Set}\omega$ falls on the author of the proof script! We manually annotated the proof script with a suitable universe level assignment to certify that our proof does not violate this constraint.

**4.4.4.** With access to the external hierarchy, we can declare external predicates as inhabitants of $A \to \mathbf{Set}_{\omega}$. Some constructions (such as the Idealization axiom and the induction principle over the natural numbers) do not accept external predicates as arguments. Fortunately, Agda knows that $\mathbf{Set}_{\omega} \neq \mathbf{Set}_{\ell}$ for any internal (i.e. actual) level $\ell$, so the Agda proof checker will automatically ensure that we do not supply an external predicate to a construction that works only with internal predicates. However, Agda's pattern matching mechanism (a shorthand notation for nested uses of induction principles) does not perform this check, and would allow us to do invalid proofs by induction,

[6] See `https://agda.readthedocs.io/en/v2.6.0.1/getting-started/tutorial-list.html`

such as proving the standardness of all natural numbers. Therefore, we have to disable definitions by pattern matching using the `-no-pattern-matching` option provided by Agda.

**4.4.5.** As discussed in Definition 4.2.35, we can introduce the Idealization and Standardization principles as proper axioms using Agda's `postulate` keyword. The same method works for encoding all TRF- rules, apart from TRF-REFL. To implement the latter, we have to introduce a distinction between *safe* and general Agda modules, in a similar way to how we separated $\vdash^s$ and $\vdash$ in Section 4.2.30. We write safe modules in the pure subset of Agda, without access to any of the previously discussed features coming from our proposed extensions. As such, a top-level definition of $t : T$ in a safe module corresponds to a derivation $\vdash^s t : T$ in our extended type theory. We add a private constructor `defined-in-safe-module` used only to declare top level definitions in safe modules standard.

**4.4.6.** Given the implementation details above, one might wonder: what do we need to believe this proof, given that Agda checked it?

1. *Martin-Löf Type Theory.* One has to believe the consistency and mathematical relevance of Martin-Löf Type Theory. As mentioned in Section 4.1.4, Martin-Löf Type Theory has been in use since the 1970s to general satisfaction and has served as a basis for many formalized proofs. The currently prevalent foundation of mathematics, Zermelo-Fraenkel Set Theory with the Axiom of Choice (along with some mild large cardinal assumptions) proves the consistency of all common variants of Martin-Löf Type Theory.

2. *The extensions to type theory.* We formalized the proof of our main result in a way that never invokes the Idealization axiom, so the proof of Proposition 4.3.15 applies and guarantees consistency.

3. *The Agda implementation.* One has to trust that the type theory implemented by the Agda proof checker accurately reflects Martin-Löf Type Theory, and our additional postulates accurately reflect the extensions. Demotically: "one has to trust that the Agda proof checker can check at least this particular proof (if nothing else)."

4. *Accurate transcription.* A formal argument establishes exactly what the author states, not necessarily what the author means, much less what the author desires.

One may look at the formalized, computer-verified proof given in the appendix and ask: "yes, you have produced a verifiable formal proof of some statement, but how do we know that the proved statement corresponds to the informal statement of Theorem 2.3.9?" We chose our proposed extensions to type theory with this requirement in mind, so that the type-theoretic development can follow the informal argument as closely as possible. Fortunately, the notions involved in the statement of our main result (groups, group actions, metric spaces, strong approximation) have short, axiomatic definitions, so one can easily verify the correspondence between the concepts and their formalized counterparts.

5. *Newman's theorem.* As discussed in the relevant chapter, Theorem 2.3.6 (Newman's theorem) provides the group-theoretic substrate of our result. In principle, one could write down and computer-verify a proof of Newman's theorem in Agda. However, a formal statement and verification of Newman's theorem lies far outside the scope of our work, and would probably make a fine research project of its own. As such, we admit Theorem 2.3.6 without proof, and rely on it as a "black box". Newman's theorem is the only such presupposition used in our argument.

**4.4.7.** The formal proof consists of approximately 3500 lines of Agda code (not counting the in-line comments), organized hierarchically into 23 modules. Table 4.1 cross-references the sections of this document with their corresponding modules. The formalized proof of Theorem 2.3.9 follows the original argument very closely, except for one minor modification. We wanted our proof to avoid appeals to Idealization, since the fragment of extended type theory in Proposition 4.3.15 omits this axiom. However, Theorem 2.3.9 depends on Proposition 1.2.14, and the textbook proof of the latter relies on an Idealization argument. We give an alternative proof (presented in Section 4.4.8) that bypasses this use of Idealization using a slightly more involved appeal to external induction.

**4.4.8** (Alternate proof of Proposition 1.2.14)**.** Consider a standard natural number $b$. All $n \in \mathbb{N}$ with $n < b$ are standard.

**Proof.** Let $\varphi(b)$ abbreviate the following property: $\forall n \in \mathbb{N}. n \leq b \to \mathrm{st}(n)$. We prove $\forall^{st} b.\varphi(b)$ using external induction (Theorem 1.2.15).

- **Base case:** We need to prove $\forall n. n \leq 0 \to \mathrm{st}(n)$. But $n \leq 0$ implies $n = 0$, and $\mathrm{st}(0)$ holds by Proposition 1.2.9.

- **Inductive case:** We have a standard $k$ such that all $n \leq k$ satisfy $\mathrm{st}(n)$. We need to prove that all $n \leq k+1$ satisfy $\mathrm{st}(n)$. If $n \leq k$, then we can conclude $\mathrm{st}(n)$ using the induction hypothesis. Otherwise, $n = k+1$, and $\mathrm{st}(k+1)$ follows from the standardness of $k$ using Corollary 1.2.11.

By the principle of external induction, we have that given any standard natural $b$, if $n \leq b$ then $\mathrm{st}(n)$.
**Qed.**

| Section | Description | Module |
|---|---|---|
| 1.2.14 | Standard naturals closed downward | `IST.Naturals` |
| 1.2.15 | External induction | `IST.Naturals` |
| 1.3.35 | Metric spaces are equivalence spaces | `IST.PredicatedTopologies` |
| 1.3.38 | Ultrafilters have monadic elements | `IST.Ultrafilters` |
| 2.3.3 | Function extension theorem | `IST.Results.ExtensionTheorem` |
| 2.3.9 | Action extension theorem | `IST.Results.MainTheorem` |

Table 4.1: Cross-reference: theorems and corresponding Agda modules.

**4.4.9.** Type-checking the proof requires Agda version 2.6.0.1. Verifying the complete proof takes less than 2 minutes on a modern computer, and needs approximately 2 gigabytes of free RAM.

$$\dfrac{\vdash^s n : \mathbb{N} \qquad \dfrac{\vdash^s n : \mathbb{N}}{\vdash^s \texttt{refl}\ \mathbb{N}\ n : n = n}\ \textsc{Eqt-Intr}}{\vdash^s (n, \texttt{refl}\ \mathbb{N}\ n) : (\exists x : \mathbb{N}.x = n)}\ \textsc{Dsum-Intr}$$
$$\cdots_1$$

$$\dfrac{\dfrac{}{\emptyset \vdash^s}\ \textsc{Ctx-Nul} \qquad \dfrac{}{\vdash^s \mathbb{N} : \mathbf{Set}_0}\ \textsc{Nat-Form}}{x : \mathbb{N} \vdash^s}\ \textsc{Ctx-Ext}$$
$$\cdots_2$$

$$\dfrac{\cdots_1 \qquad \dfrac{\dfrac{\vdash^s n : \mathbb{N} \qquad \dfrac{\dfrac{\dfrac{}{x : \mathbb{N} \vdash^s \mathbb{N} : \mathbf{Set}_0}\ \textsc{Nat-Form} \qquad \dfrac{\cdots_2}{x : \mathbb{N} \vdash^s x : \mathbb{N}}\ \textsc{Var} \qquad x : \mathbb{N} \vdash^s n : \mathbb{N}}{x : \mathbb{N} \vdash x =_{\mathbb{N}} n : \mathbf{Set}_0}\ \textsc{Eqt-Form}}{x : \mathbb{N} \vdash x = n \Leftrightarrow_0 x = n}\ \star\ \textsc{Trf-Refl}}{\vdash \exists x : \mathbb{N}.x = n \Leftrightarrow_0 \exists^{st} x : \mathbb{N}.x = n}\ \star\ \textsc{Trf-Dsum} \qquad \dfrac{}{\emptyset \vdash}\ \textsc{Ctx-Nul}}{\vdash \texttt{TraL} : \left(\exists x : \mathbb{N}.x = n \to \exists^{st} x : \mathbb{N}.x = n\right)}\ \star\ \textsc{Ax-TraL}}{\vdash \texttt{TraL}\ (n, \texttt{refl}\ \mathbb{N}\ n) : \left(\exists^{st} x : \mathbb{N}.x = n\right)}\ \textsc{Dfun-Elim}$$

Figure 4.1: A closed term of type $\exists x : \mathbb{N}.\texttt{st}_0\ \mathbb{N}\ x \times x =_{\mathbb{N}} n$ for each canonical natural $n : \mathbb{N}$.

$$\dfrac{\dfrac{}{\emptyset \vdash}\ \textsc{Ctx-Nul} \qquad \dfrac{\dfrac{}{\vdash \mathbb{N} : \mathbf{Set}_0}\ \textsc{Nat-Form} \qquad \vdash \pi_1 f' : \mathbb{N}}{\vdash \mathtt{st}_0\,\mathbb{N}\,(\pi_1 f') : \mathbf{Set}_\omega}\ \star\ \textsc{St-Form}}{p : \boldsymbol{P} \vdash}\ \textsc{Ctx-Ext}$$

$$\cdots_2$$

$$\dfrac{\dfrac{p : \boldsymbol{P} \vdash \pi_2 f' : \forall^{st} y : \mathbb{N}.\mathtt{st}\,\mathbb{N}\,y \to \pi_1 f' =_{\mathbb{N}} y \to \bot \qquad p : \boldsymbol{P} \vdash \pi_1 f' : \mathbb{N}}{p : \boldsymbol{P} \vdash \pi_2 f'\,(\pi_1 f') : \boldsymbol{P} \to \pi_1 f' =_{\mathbb{N}} \pi_1 f' \to \bot}\ \textsc{Dfun-Elim} \qquad \dfrac{\cdots_2}{p : \boldsymbol{P} \vdash p : \boldsymbol{P}}\ \textsc{Var}}{p : \boldsymbol{P} \vdash \pi_2 f'\,(\pi_1 f')\,p : \boldsymbol{P} \to \bot}\ \textsc{Dfun-Elim}$$

$$\cdots_1$$

$$\dfrac{\dfrac{\cdots_1 \qquad \dfrac{p : \boldsymbol{P} \vdash \pi_1 f' : \mathbb{N}}{p : \boldsymbol{P} \vdash \mathtt{refl}\,\mathbb{N}\,(\pi_1 f') : \pi_1 f' =_{\mathbb{N}} \pi_1 f'}\ \textsc{Eqt-Intr}}{p : \boldsymbol{P} \vdash \pi_2 f'\,(\pi_1 f')\,p\,(\mathtt{refl}\,\mathbb{N}\,(\pi_1 f')) : \bot}\ \textsc{Dfun-Elim}}{\vdash (\lambda p.\pi_2 f'\,(\pi_1 f')\,p\,(\mathtt{refl}\,\mathbb{N}\,(\pi_1 f'))) : \mathtt{st}_0\,\mathbb{N}\,(\pi_1 f') \to \bot}\ \textsc{Dfun-Intr}$$

where $\boldsymbol{P}$ abbreviates the term $\mathtt{st}_0\,\mathbb{N}\,(\pi_1 f')$.

Figure 4.2: Derivation tree witnessing the existence of a nonstandard number.

# Bibliography


[1] M. A. ALEKSEEV, L. Y. GLEBSKII, AND E. I. GORDON, *On approximation of groups, group actions, and Hopf algebras*, Journal of Mathematical Sciences, 107 (2001), pp. 4305–4332.

[2] M. ARBO, K. BENKOWSKI, B. COATE, H. NORDSTROM, C. PETERSON, AND A. WOOTTON, *The genus level of a group*, Involve, 2 (2009), pp. 323–340.

[3] L. BABAI, K. FRIEDL, AND A. LUKÁCS, *Near-representations of finite groups*, tech. rep., Computer and Automation Research Institute, Hungarian Academy of Sciences, 06 2003.

[4] A. BLASS, *A model without ultrafilters*, Bulletin of the Polish Academy of Sciences: Mathematics, Astronomy, Physics, 4 (1977), pp. 329–331.

[5] P. BASZCZYK, V. KANOVEI, AND M. KATZ, *Monotone subsequence via Ultrapower*, Open Mathematics, 16 (2018), pp. 149–153.

[6] T. CECCHERINI-SILBERSTEIN, *Cellular automata and groups*, Springer-Verlag, Heidelberg New York, 2010.

[7] T. CECCHERINI-SILBERSTEIN AND M. COORNAERT, *Residually finite groups*, in Cellular Automata and Groups, T. Ceccherini-Silberstein, ed., Springer Berlin Heidelberg, Berlin, Heidelberg, 2010, pp. 37–55.

[8] G. CHERLIN AND J. HIRSCHFELD, *Ultrafilters and ultraproducts in non-standard analysis*, in Contributions to Non-Standard Analysis, W. Luxemburg and A. Robinson, eds., vol. 69 of Studies in Logic and the Foundations of Mathematics, Elsevier, 1972, pp. 261 – 279.

[9] T. COQUAND AND B. MANNAA, *The Independence of Markov's Principle in Type Theory*, in 1st International Conference on Formal Structures for Computation and Deduction (FSCD 2016), D. Kesner and B. Pientka, eds., vol. 52 of

Leibniz International Proceedings in Informatics (LIPIcs), Dagstuhl, Germany, 2016, Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, pp. 17:1–17:18.

[10] N. A. DANIELSSON, *Papers using Agda*. The Agda Wiki - `https://wiki.portal.chalmers.se/agda/pmwiki.php?n=Main.PapersUsingAgda`. Accessed: 2019-07-01.

[11] N. G. DE BRUIJN, *Automath, a language for mathematics*, in Automation of Reasoning: 2: Classical Papers on Computational Logic 1967–1970, J. H. Siekmann and G. Wrightson, eds., Springer Berlin Heidelberg, Berlin, Heidelberg, 1983, pp. 159–200.

[12] F. DIENER AND M. DIENER, eds., *Nonstandard Analysis in Practice*, Springer Berlin Heidelberg, 1995.

[13] M. DOUCHA, *Metric topological groups: their metric approximation and metric ultraproducts*, to appear in Groups, Geometry, and Dynamics, (2016), p. arXiv:1601.07449.

[14] P. GARDNER, *Representing Logics in Type Theory*, PhD thesis, University of Edinburgh, UK, 1992.

[15] J.-Y. GIRARD, P. TAYLOR, AND Y. LAFONT, *Proofs and Types*, Cambridge University Press, New York, NY, USA, 1989.

[16] R. GOLDBLATT, *Lectures on the Hyperreals: an Introduction to Nonstandard Analysis*, Springer, New York, 1998.

[17] G. GONTHIER, *The four colour theorem: Engineering of a formal proof*, in Computer Mathematics, D. Kapur, ed., Berlin, Heidelberg, 2008, Springer Berlin Heidelberg, p. 333.

[18] G. GONTHIER, A. ASPERTI, J. AVIGAD, Y. BERTOT, C. COHEN, F. GARILLOT, S. LE ROUX, A. MAHBOUBI, R. O'CONNOR, S. OULD BIHA, I. PASCA, L. RIDEAU, A. SOLOVYEV, E. TASSI, AND L. THÉRY, *A Machine-Checked Proof of the Odd Order Theorem*, in ITP 2013, 4th Conference on Interactive Theorem Proving, S. Blazy, C. Paulin, and D. Pichardie, eds., vol. 7998 of LNCS, Rennes, France, July 2013, Springer, pp. 163–179.

[19] E. GUNTHER, A. GADEA, AND M. PAGANO, *Formalization of universal algebra in Agda*, Electronic Notes in Theoretical Computer Science, 338 (2018), pp. 147 – 166. The 12th Workshop on Logical and Semantic Frameworks, with Applications (LSFA 2017).

[20] M. HENLE, *A Combinatorial Introduction to Topology*, Dover Publications, 1994.

[21] C. HERMIDA, U. S. REDDY, AND E. P. ROBINSON, *Logical relations and parametricity - a Reynolds Programme for Category Theory and Programming Languages*, Electronic Notes in Theoretical Computer Science, 303 (2014), pp. 149 – 180. Proceedings of the Workshop on Algebra, Coalgebra and Topology 2013.

[22] M. HOFMANN, *Syntax and semantics of dependent types*, in Semantics and Logics of Computation, Cambridge University Press, 1997, pp. 79–130.

[23] K. HRBACEK, *Axiom of choice in nonstandard set theory*, Journal of Logic and Analysis [electronic only], 4 (2012).

[24] A. J. C. HURKENS, *A simplification of girard's paradox*, in Typed Lambda Calculi and Applications, M. Dezani-Ciancaglini and G. Plotkin, eds., Berlin, Heidelberg, 1995, Springer Berlin Heidelberg, pp. 266–278.

[25] T. IMAMURA, *A nonstandard construction of direct limit group actions*, arXiv e-prints, (2018), p. arXiv:1812.00575.

[26] V. KANOVEI AND M. REEKEN, *Nonstandard Analysis, Axiomatically*, Springer Monographs in Mathematics, Springer Berlin Heidelberg, 2013.

[27] M. M. KAPRANOV AND V. A. VOEVODSKY, *Infinity-groupoids and homotopy types*, Cahiers de Topologie et Géométrie Différentielle Catégoriques, 32 (1991), pp. 29–46.

[28] N. KRISHNASWAMI, *Explicit set of types and terms in Martin-Loef type theory*. Theoretical Computer Science Stack Exchange. - `https://cstheory.stackexchange.com/q/39361` (version: 2017-10-24).

[29] A. LEVY, *Basic Set Theory*, Dover Publications, Mineola, NY, USA, 2002.

[30] L. M. MANEVITZ AND S. WEINBERGER, *Discrete circle actions: A note using non-standard analysis*, Israel Journal of Mathematics, 94 (1996), pp. 147–155.

[31] J. A. C. MORALES AND B. ZILBER, *The geometric semantics of algebraic quantum mechanics*, Philosophical Transactions of the Royal Society A: Mathematical, Physical and Engineering Sciences, 373 (2015).

[32] E. NELSON, *Internal set theory: A new approach to nonstandard analysis*, Bulletin of the American Mathematical Society, 83 (1977), pp. 1165–1198.

[33] E. NELSON, *The syntax of nonstandard analysis*, Annals of Pure and Applied Logic, 38 (1988), pp. 123 – 134.

[34] N. NIKOLOV, J. SCHNEIDER, AND A. THOM, *Some remarks on finitarily approximable groups*, Journal de l'Ecole Polytechnique - Mathematiques, 5 (2017).

[35] U. NORELL, *Towards a practical programming language based on dependent type theory*, PhD thesis, Department of Computer Science and Engineering, Chalmers University of Technology, SE-412 96 Gothenburg, Sweden, September 2007.

[36] U. NORELL, *Dependently Typed Programming in Agda*, Springer Berlin Heidelberg, Berlin, Heidelberg, 2009, pp. 230–266.

[37] J. PARDON, *Totally disconnected groups (not) acting on two-manifolds*, arXiv e-prints, (2018), p. arXiv:1811.08748.

[38] A. PILLAY, *Remarks on compactifications of pseudofinite groups*, Fundamenta Mathematicae, 236 (2017), pp. 193 – 200.

[39] D. R. LICATA AND M. SHULMAN, *Calculating the fundamental group of the circle in Homotopy Type Theory*, in Proceedings of the Symposium on Logic in Computer Science, 06 2013, pp. 223–232.

[40] A. ROBERT, *Nonstandard analysis*, Wiley-Interscience Publications, Wiley, 1988.

[41] S. SANDERS, *The unreasonable effectiveness of Nonstandard Analysis*, arXiv e-prints, (2015), p. arXiv:1508.07434.

[42] ——, *A note on non-classical Nonstandard Arithmetic*, to appear in Annals of Pure and Applied Logic, (2018), p. arXiv:1805.11705.

[43] G. SCHERER AND A. ABEL, *Universe subtyping in Martin-Löf type theory (internship report*, tech. rep., Department of Computer Science and Engineering, Chalmers University of Technology, 2011.

[44] C. SIMPSON, *Homotopy Theory of Higher Categories: From Segal Categories to n-Categories and Beyond*, New Mathematical Monographs, Cambridge University Press, 2011.

[45] THE UNIVALENT FOUNDATIONS PROGRAM, *Homotopy Type Theory: Univalent foundations of mathematics*, tech. rep., Institute for Advanced Study, 2013.

[46] B. VAN DEN BERG, E. BRISEID, AND P. SAFARIK, *A functional interpretation for nonstandard arithmetic*, Annals of Pure and Applied Logic, 163 (2012), pp. 1962 – 1994.

[47] S. VICKERS, *Fuzzy sets and geometric logic*, Fuzzy Sets and Systems, 161 (2010), pp. 1175 – 1204. Foundations of Lattice-Valued Mathematics with Applications to Algebra and Topology.

[48] V. A. VOEVODSKY, *Special year on Univalent Foundations of Mathematics*. Programme Proposal, Institute for Advanced Study, 2012.

[49] C. XU, *Implementing the nonstandard Dialectica interpretation*. Github - `https://cj-xu.github.io/agda/nonstandard_dialectica/Index.html`. Accessed: 2019-07-01.

[50] B. ZILBER, *Structural approximation*. draft on author's personal website (https://people.maths.ox.ac.uk/zilber/approx.pdf), Mar 2010.

[51] B. ZILBER, *Perfect Infinities and Finite Approximation*, in Infinity and Truth, C. Chitat, Q. Feng, T. A. Slaman, and W. H. Woodin, eds., WSPC, 2013.

# Appendix A

# Agda Proof of Theorem 2.3.9

The next 52 pages contain the 2019-08-27 revision of the Agda proofs described in Chapter 4. The newest version of the proof is available in the Github repository at

`https://github.com/zaklogician/agda-ist-algebra.git`

```
module IST.Safe.Base where

-- Here we define standard constructions from type theory, including
-- the usual dependent sum types and equality. This development
-- takes place in ordinary MLTT/Agda, without the external hierarchy.

open import Agda.Primitive


-- TRIVIAL DATA TYPES --

-- The empty type and ex falso quodlibet.

data ⊥ : Set where

absurd : {ℓ : Level} → ⊥ → ∀ {A : Set ℓ} → A
absurd ()

-- The singleton type.

data ⊤ : Set where
  tt : ⊤


-- EXISTENTIAL QUANTIFICATION --

-- Now we deal with existential quantifiers. Alas, unlike the ∀
-- case, Agda does not provide a builtin for this, so we need to
-- declare two variants, ∃ (for the internal hierarchy) and ∃*
-- (for the external hierarchy). Here we declare the internal
-- variant ∃, and define ∧ in terms of it.

infixr 4 _,_
record ∃ {ℓ₁ ℓ₂ : Level} {A : Set ℓ₁} (B : A → Set ℓ₂) : Set (ℓ₂ ⊔ ℓ₁) where
  constructor _,_
  field
    proj₁ : A
    proj₂ : B proj₁
open ∃ public

_∧_ : ∀ {ℓ₁ ℓ₂} → Set ℓ₁ → Set ℓ₂ → Set (ℓ₂ ⊔ ℓ₁)
A ∧ B = ∃ λ (x : A) → B


-- LISTS / FINITE SETS --

data List {ℓ : Level} (A : Set ℓ) : Set ℓ where
  [] : List A
  _∷_ : A → (xs : List A) → List A

List-induction : {ℓ₁ ℓ₂ : Level} {A : Set ℓ₁} {B : List A → Set ℓ₂} →
                 B [] → (∀ x → ∀ xs → B xs → B (x ∷ xs)) → ∀ y → B y
List-induction base-case inductive-case [] = base-case
List-induction base-case inductive-case (x ∷ xs) =
  inductive-case x xs (List-induction base-case inductive-case xs)

data _∈_ {ℓ} {A : Set ℓ} (x : A) : List A → Set ℓ where
  ∈-head : ∀ {ys} → x ∈ (x ∷ ys)
  ∈-tail : ∀ {y ys} → x ∈ ys → x ∈ (y ∷ ys)


-- DISJUNCTION --

-- We could encode (constructive) disjunction using ∃ and a two-element
-- type, but declaring an explicit data type keeps reasoning much
-- more legible.

data _∨_ {ℓ₁ ℓ₂ : Level} (A : Set ℓ₁) (B : Set ℓ₂) : Set (ℓ₁ ⊔ ℓ₂) where
  inl : A → A ∨ B
  inr : B → A ∨ B

by-cases : {ℓ₁ ℓ₂ ℓ₃ : Level} {A : Set ℓ₁} {B : Set ℓ₂} →
           (P : Set ℓ₃) → (A → P) → (B → P) → A ∨ B → P
by-cases P A-implies-P B-implies-P (inl a) = A-implies-P a
```

```
by-cases P A-implies-P B-implies-P (inr b) = B-implies-P b

postulate
  by-LEM : {ℓ₁ ℓ₂ : Level} → {A : Set ℓ₂} → {B : Set ℓ₂} → ((A → ⊥) → B) → A ∨ B


-- EQUALITY --

-- We define equality only for the internal hierarchy, but only
-- with the internal induction principle. Later on,
-- we will admit a transport* principle for the external
-- hierarchy. This counts as a "hidden axiom" of IST, because
-- first-order logic is assumed to have equality. Really, we'd need
-- to check that the Nelson translation of (x = y) → st(x) → st(y) is
-- provable in ZFC.

infix 4 _≡_
data _≡_ {ℓ : Level} {A : Set ℓ} (x : A) : A → Set where
  refl : x ≡ x

sym : {ℓ : Level} {A : Set ℓ} {x y : A} → x ≡ y → y ≡ x
sym refl = refl

tran : {ℓ : Level} {A : Set ℓ} {x y z : A} → x ≡ y → y ≡ z → x ≡ z
tran refl refl = refl

cong : {ℓ : Level} {A B : Set ℓ} {x y : A} → (f : A → B) → x ≡ y → f x ≡ f y
cong f refl = refl

transport : {ℓ₁ ℓ₂ : Level} {A : Set ℓ₁} {x y : A} → x ≡ y → ∀ {φ : A → Set ℓ₂} → φ x → φ y
transport refl z = z

≡-ind : {ℓ₁ ℓ₂ : Level} → {A : Set ℓ₁} → {x : A} → (φ : (x ≡ x) → Set ℓ₂) → φ refl → ∀ e → φ e
≡-ind φ p refl = p


-- COMBINATORIAL (CLOSED) LAMBDA TERMS --

-- We cannot use induction on Set-types, so how do we prove them
-- standard? In IST, we do not have to deal with this problem,
-- since we normally encode functions as their graphs (sets of
-- ordered pairs), and IST already provides rules for the
-- standardness of sets.

-- In Agda, functions do not coincide with sets of ordered pairs,
-- and we need to ensure that all MLTT-definable functions are
-- indeed standard. To accomplish this, the rules below suffice:
-- 1. All pure combinatorial λ-terms with standard co/domain are themselves standard.
-- 2. Functions defined by induction are standard.
-- 3. Applying a standard value to a standard function yields a standard result.
-- These rules exhaust all possible ways of defining functions
-- in MLTT.

-- E.g. to prove that (λi. _=_ (f i) (g i)) is standard, we would
-- argue as follows:
-- 1. (\a.\b.\c. a b c) is a purely combinatorial λ-term, so standard.
-- 2. (\b.\c _=_ b c) is standard when both _=_ and (\a.\b.\c. a b c) are standard.
-- 3. (\c _=_ (f i) c) is standard when both (f i) and (\b.\c _=_ b c) are standard.
-- 4. (_=_ (f i) (g i)) is standard when both (g i) and (\c. _=_ (f i) (g i)) are standard.
-- So we'd conclude that the inhabitant (λi. _=_ (f i) (g i))
-- of the type Set is standard as long as (f i) and (g i) are.

-- Here we declare the combinatorial instances that we actually use
-- in our development, so that we can safely declare them standard in
-- IST.Base.

abs-5 : (I : Set) → ({X : Set} → X → X → Set) →
        (b : I → Set) →
        (f : (i : I) → b i) →
        (g : (i : I) → b i) →
        (i : I) → Set
abs-5 I = λ (a : {X : Set} → X → X → Set) →
          λ (b : I → Set) →
          λ (f : (i : I) → b i) →
          λ (g : (i : I) → b i) →
          λ (i : I) → a {b i} (f i) (g i)
```

```
abs-4 : (A M X : Set) →
        (f : A → M → M) →
        (e : X → A) →
        X → M → M
abs-4 A M X f e x m = f (e x) m

abs-K : {ℓ₁ ℓ₂ : Level} (A : Set ℓ₁) (B : Set ℓ₂) → A → B → A
abs-K A B = λ (a : A) → λ (b : B) → a

abs-K-h : {ℓ₁ ℓ₂ : Level} (A : Set ℓ₁) (B : Set ℓ₂) → A → {_ : B} → A
abs-K-h A B = λ (a : A) → λ {b : B} → a


-- We admit the law of excluded middle.

postulate
  excluded-middle : {ℓ : Level} → (A : Set ℓ) → A ∨ (A → ⊥)

--------------------------------------------------------------------------------


module IST.Safe.Util where

-- Here we define standard constructions from type theory, including
-- the usual dependent sum types and equality. This development
-- takes place in ordinary MLTT/Agda, without the external hierarchy.

open import Agda.Primitive
open import IST.Safe.Base

lemma-product-equality : {ℓ₁ ℓ₂ : Level} {X : Set ℓ₁} {Y : Set ℓ₂} {x₁ x₂ : X} → ∀ {y₁ y₂ : Y} →
                         x₁ ≡ x₂ → y₁ ≡ y₂ → (x₁ , y₁) ≡ (x₂ , y₂)
lemma-product-equality refl refl = refl

--------------------------------------------------------------------------------


module IST.Safe.Naturals where

open import Agda.Primitive
open import IST.Safe.Base

data ℕ : Set where
  zero : ℕ
  suc : ℕ → ℕ

{-# BUILTIN NATURAL ℕ #-}

ℕ-induction : {ℓ : Level} → {φ : ℕ → Set ℓ} →
              φ 0 → (∀ k → φ k → φ (suc k)) → ∀ n → φ n
ℕ-induction base-case inductive-case zero = base-case
ℕ-induction base-case inductive-case (suc n) = inductive-case n (ℕ-induction base-case
inductive-case n)

data _≤_ : ℕ → ℕ → Set where
  ≤-zero : {x : ℕ} → 0 ≤ x
  ≤-suc : {x y : ℕ} → x ≤ y → suc x ≤ suc y

≤-tran : (x y z : ℕ) → x ≤ y → y ≤ z → x ≤ z
≤-tran .0 y z ≤-zero q = ≤-zero
≤-tran .(suc _) .(suc _) .(suc _) (≤-suc p) (≤-suc q) = ≤-suc (≤-tran _ _ _ p q)

≤-than-zero : (x : ℕ) → x ≤ 0 → x ≡ 0
≤-than-zero .0 ≤-zero = refl

≤-refl : ∀ x → x ≤ x
≤-refl zero = ≤-zero
≤-refl (suc x) = ≤-suc (≤-refl x)

≤-not-suc : ∀ x → suc x ≤ x → ⊥
≤-not-suc zero ()
≤-not-suc (suc x) (≤-suc p) = ≤-not-suc x p
```

```
≤-match : (x y : ℕ) → x ≤ suc y → (x ≤ y) ∨ (x ≡ suc y)
≤-match .0 y ≤-zero = inl ≤-zero
≤-match (suc a) zero (≤-suc p) = inr (cong suc (≤-than-zero a p))
≤-match (suc a) (suc b) (≤-suc p) with ≤-match a b p
≤-match (suc a) (suc b) (≤-suc p) | inl q = inl (≤-suc q)
≤-match (suc a) (suc b) (≤-suc p) | inr q = inr (cong suc q)

--------------------------------------------------------------------------------


module IST.Safe.FiniteSets where

open import IST.Safe.Base

record IsFiniteSet
  (Carrier : Set)
  : Set where
  field
    list-of-elements : List Carrier
    has-all-elements : (x : Carrier) → x ∈ list-of-elements

record FiniteSet : Set₁ where
  field
    Carrier : Set
    isFiniteSet : IsFiniteSet Carrier
  open IsFiniteSet isFiniteSet public

--------------------------------------------------------------------------------


module IST.Safe.Reals where

open import Agda.Primitive
open import IST.Safe.Base

-- We present an ordered field axiomatically. We do not give a completeness axiom.

-- ℝ forms a commutative ring.
infixr 5 _+_
infixr 6 _·_
postulate
  ℝ : Set
  _+_ : ℝ → ℝ → ℝ
  0r : ℝ
  +-comm : ∀ {x y : ℝ} → x + y ≡ y + x
  +-assoc : ∀ {x y z : ℝ} → (x + y) + z ≡ x + (y + z)
  +-unit-left : ∀ {x : ℝ} → 0r + x ≡ x
  minus : ℝ → ℝ
  +-inverse-left : ∀ {x : ℝ} → x + minus x ≡ 0r

  _·_ : ℝ → ℝ → ℝ
  1r : ℝ
  ·-comm : ∀ {x y : ℝ} → x · y ≡ y · x
  ·-assoc : ∀ {x y z : ℝ} → (x · y) · z ≡ x · (y · z)
  ·-unit-left : ∀ {x : ℝ} → 1r · x ≡ x
  ·-null-left : ∀ {x : ℝ} → x · 0r ≡ 0r

  distr-left : ∀ {x y z : ℝ} → x · (y + z) ≡ x · y + x · z

-- The right laws follow by commutativity.

+-unit-right : ∀ {x : ℝ} → x + 0r ≡ x
+-unit-right = tran +-comm +-unit-left

·-unit-right : ∀ {x : ℝ} → x · 1r ≡ x
·-unit-right = tran ·-comm ·-unit-left

·-null-right : ∀ {x : ℝ} → 0r · x ≡ 0r
·-null-right = tran ·-comm ·-null-left

distr-right : ∀ {x y z : ℝ} → (x + y) · z ≡ x · z + y · z
distr-right {x} {y} {z} = step-3 where
  step-1 : z · x + z · y ≡ x · z + z · y
  step-1 = cong (λ p → p + z · y) ·-comm
```

```
  step-2 : x · z + z · y ≡ x · z + y · z
  step-2 = cong (λ p → x · z + p) ·-comm
  step-3 : (x + y) · z ≡ x · z + y · z
  step-3 = tran (tran (tran ·-comm distr-left) step-1) step-2

-- ℝ forms an ordered commutative ring.

infix 4 _<_
infix 4 _≤ᵣ_
postulate
  _<_ : ℝ → ℝ → Set
  <-trichotomy-strong : ∀ x y → (x ≡ y) ∨ ((x < y) ∨ (y < x))
  <-asym-1 : ∀ x y → x < y → x ≡ y → ⊥
  <-asym-2 : ∀ x y → x < y → y < x → ⊥
  <-tran : ∀ x y z → x < y → y < z → x < z
  <-plus : ∀ x y c → x < y → x + c < y + c
  <-mult : ∀ x y c → 0r < c → x < y → c · x < c · y
  <-nontrivial : 0r < 1r

-- ℝ forms a field

_≠_ : ℝ → ℝ → Set
x ≠ y = ((x < y) → ⊥) → y < x

postulate
  inv : (r : ℝ) → (r ≠ 0r) → ℝ
  ·-inverse-left : ∀ {x : ℝ} → (p : x ≠ 0r) → inv x p · x ≡ 1r

+-inverse-right : ∀ {x : ℝ} → minus x + x ≡ 0r
+-inverse-right = tran +-comm +-inverse-left

·-inverse-right : ∀ {x : ℝ} → (x≠0 : x ≠ 0r) → x · inv x x≠0 ≡ 1r
·-inverse-right x≠0 = tran ·-comm (·-inverse-left x≠0)

-- We state and prove some useful elementary theorems about ℝ.

·-minus : ∀ {x : ℝ} → minus x ≡ (minus 1r) · x
·-minus {x} = sym step-9 where
  step-1 : x + minus 1r · x ≡ (1r · x) + minus 1r · x
  step-1 = cong (λ p → p + minus 1r · x) (sym (·-unit-left))
  step-2 : 1r · x + minus 1r · x ≡ (1r + minus 1r) · x
  step-2 = sym (distr-right)
  step-3 : (1r + minus 1r) · x ≡ 0r
  step-3 = tran (cong (λ p → p · x) +-inverse-left) ·-null-right
  step-4 : x + minus 1r · x ≡ 0r
  step-4 = tran (tran step-1 step-2) step-3
  step-5 : minus x + (x + minus 1r · x) ≡ minus x + 0r
  step-5 = cong (λ p → minus x + p) step-4
  step-6 : minus x + (x + minus 1r · x) ≡ minus x
  step-6 = tran step-5 +-unit-right
  step-7 : (minus x + x) + minus 1r · x ≡ minus x
  step-7 = tran +-assoc step-6
  step-8 : 0r + minus 1r · x ≡ minus x
  step-8 = tran (cong (λ p → p + minus 1r · x) (sym +-inverse-right)) step-7
  step-9 : minus 1r · x ≡ minus x
  step-9 = tran (sym +-unit-left) step-8

<-trichotomy : ∀ {φ : Set} → ∀ x y → (x < y → φ) → (x ≡ y → φ) → (y < x → φ) → φ
<-trichotomy {φ} x y p q r with <-trichotomy-strong x y
<-trichotomy {φ} x y p q r | inl x-equals-y = q x-equals-y
<-trichotomy {φ} x y p q r | inr (inl x-under-y) = p x-under-y
<-trichotomy {φ} x y p q r | inr (inr y-under-x) = r y-under-x

<-minus : ∀ {x : ℝ} → 0r < x → minus x < 0r
<-minus {x} positive-x = <-trichotomy 0r (minus x)
  (λ z → absurd (not-positive z))
  (λ z → absurd (not-zero z))
  (λ z → z) where
  not-positive : 0r < minus x → ⊥
  not-positive positive-minus-x = <-asym-2 0r x positive-x negative-x  where
    step-1 : 0r + x < minus x + x
    step-1 = <-plus 0r (minus x) x positive-minus-x
    step-2 : 0r + x < 0r
    step-2 = transport +-inverse-right {λ p → 0r + x < p} step-1
```

```
    negative-x : x < 0r
    negative-x = transport +-unit-left {λ p → p < 0r} step-2
  not-zero : 0r ≡ minus x → ⊥
  not-zero zero-minus-x = <-asym-1 0r x positive-x (sym zero-x) where
    step-1 : x + 0r ≡ 0r
    step-1 = transport (sym zero-minus-x) {λ p → x + p ≡ 0r} +-inverse-left
    zero-x : x ≡ 0r
    zero-x = transport (+-unit-right {x}) {λ p → p ≡ 0r} step-1

<-inverse : ∀ {x : ℝ} → (p : 0r < x) → 0r < inv x (λ _ → p)
<-inverse {x} p = <-trichotomy 0r (inv x (λ _ → p))
  (λ z → z)
  (λ z → absurd (not-zero z))
  (λ z → absurd (not-negative z)) where
  not-zero : 0r ≡ inv x (λ _ → p) → ⊥
  not-zero 0r-inverse = <-asym-1 _ _ <-nontrivial step-3 where
    step-1 : 0r · x ≡ inv x (λ _ → p) · x
    step-1 = cong (λ p → p · x) 0r-inverse
    step-2 : 0r · x ≡ 1r
    step-2 = tran step-1 (·-inverse-left (λ _ → p))
    step-3 : 0r ≡ 1r
    step-3 = tran (sym ·-null-right) step-2
  not-negative : inv x (λ _ → p) < 0r → ⊥
  not-negative negative-invx = <-asym-2 _ _ <-nontrivial step-3 where
    x' : ℝ
    x' = inv x (λ _ → p)
    step-1 : x · x' < x · 0r
    step-1 = <-mult x' 0r x p negative-invx
    step-2 : 1r < x · 0r
    step-2 = transport (·-inverse-right (λ _ → p)) {λ p → p < x · 0r} step-1
    step-3 : 1r < 0r
    step-3 = transport ·-null-left {λ p → 1r < p} step-2

<-plus-both : ∀ (x X y Y : ℝ) → x < X → y < Y → x + y < X + Y
<-plus-both x X y Y p q = <-tran _ _ _ step-1 step-4 where
  step-1 : x + y < X + y
  step-1 = <-plus x X y p
  step-2 : y + X < Y + X
  step-2 = <-plus y Y X q
  step-3 : X + y < Y + X
  step-3 = transport (+-comm {y} {X}) {λ z → z < Y + X} step-2
  step-4 : X + y < X + Y
  step-4 = transport (+-comm {Y} {X}) {λ z → X + y < z} step-3

<-plus-left : ∀ x y c → x < y → (c + x) < (c + y)
<-plus-left x y c p = step-3 where
  step-1 : x + c < y + c
  step-1 = <-plus x y c p
  step-2 : c + x < y + c
  step-2 = transport +-comm {λ p → p < y + c} step-1
  step-3 : c + x < c + y
  step-3 = transport +-comm {λ p → c + x < p} step-2

lemma-ε-of-room : ∀ (x : ℝ) → (∀ (ε : ℝ) → 0r < ε → x < ε) → (x < 0r → ⊥) → x ≡ 0r
lemma-ε-of-room x x<ε x≥0 = <-trichotomy {x ≡ 0r} x 0r
  (λ z → absurd (x≥0 z))
  (λ z → z)
  (λ z → absurd (<-asym-2 x x (x<ε x z) (x<ε x z)))

lemma-ε-of-room-plus : ∀ (x y : ℝ) → (∀ (ε : ℝ) → 0r < ε → x < y + ε) → (x ≡ y → ⊥) → x < y
lemma-ε-of-room-plus x y x<ε x≥y = <-trichotomy {x < y} x y
  (λ z → z)
  (λ z → absurd (x≥y z))
  (λ z → x<y z) where
    0<x-y : y < x → 0r < x + minus y
    0<x-y y<x = absurd (<-asym-1 x x x<x refl) where
      step-1 : y + minus y < x + minus y
      step-1 = <-plus y x (minus y) y<x
      step-2 : 0r < x + minus y
      step-2 = transport +-inverse-left {λ z → z < x + minus y} step-1
      step-3 : x < y + x + minus y
      step-3 = x<ε (x + minus y) step-2
      step-4 : y + x + minus y ≡ y + minus y + x
      step-4 = cong (λ z → y + z) (+-comm {x} {minus y})
      step-5 : (y + minus y) + x ≡ x
      step-5 = tran (cong (λ z → z + x) +-inverse-left) +-unit-left
```

```
      step-6 : y + x + minus y ≡ x
      step-6 = tran step-4 (tran (sym +-assoc) step-5)
      x<x : x < x
      x<x = transport step-6 {λ z → x < z} step-3
    x<y : y < x → x < y
    x<y y<x = absurd (<-asym-1 x x x<x refl) where
      x<y+x-y : x < y + x + minus y
      x<y+x-y = x<ε (x + minus y) (0<x-y y<x)
      y+x-y-equals-x : y + x + minus y ≡ x
      y+x-y-equals-x =
        tran (cong (λ z → y + z) +-comm)
        (tran (sym +-assoc) (tran (cong (λ z → z + x) +-inverse-left) +-unit-left))
      x<x : x < x
      x<x = transport y+x-y-equals-x {λ z → x < z} x<y+x-y

-- We prove some theorems about 1/2 that we need to work with metric spaces.

2r : ℝ
2r = 1r + 1r

pos-2r : 0r < 2r
pos-2r = <-tran 0r 1r 2r <-nontrivial step-2 where
  step-1 : 0r + 1r < 1r + 1r
  step-1 = <-plus 0r 1r 1r <-nontrivial
  step-2 : 1r < 1r + 1r
  step-2 = transport +-unit-left {λ p → p < 1r + 1r} step-1

1r-less-than-2r : 1r < 2r
1r-less-than-2r = step-2 where
  step-1 : 0r + 1r < 1r + 1r
  step-1 = <-plus 0r 1r 1r <-nontrivial
  step-2 : 1r < 1r + 1r
  step-2 = transport +-unit-left {λ p → p < 1r + 1r} step-1

1/2r : ℝ
1/2r = inv 2r (λ _ → pos-2r)

pos-1/2r : 0r < 1/2r
pos-1/2r = <-inverse pos-2r

1/2r-less-than-1r : 1/2r < 1r
1/2r-less-than-1r = step-3 where
  step-1 : 1/2r · 1r < 1/2r · 2r
  step-1 = <-mult 1r 2r 1/2r pos-1/2r 1r-less-than-2r
  step-2 : 1/2r < 1/2r · 2r
  step-2 = transport ·-unit-right {λ p → p < 1/2r · 2r} step-1
  step-3 : 1/2r < 1r
  step-3 = transport (·-inverse-left (λ _ → pos-2r)) {λ p → 1/2r < p} step-2

1/2r-half : 1/2r + 1/2r ≡ 1r
1/2r-half = tran step-6 (tran step-5 (tran step-4 step-3)) where
  step-1 : 2r · (1/2r + 1/2r) ≡ 2r · 1/2r + 2r · 1/2r
  step-1 = distr-left {2r} {1/2r} {1/2r}
  step-2 : 2r · 1/2r + 2r · 1/2r ≡ 2r
  step-2 = cong (λ p → p + p) (·-inverse-right (λ _ → pos-2r))
  step-3 : 1/2r · 2r · (1/2r + 1/2r) ≡ 1r
  step-3 = tran (cong (λ p → 1/2r · p) (tran step-1 step-2)) (·-inverse-left (λ _ → pos-2r))
  step-4 : (1/2r · 2r) · (1/2r + 1/2r) ≡ 1/2r · 2r · (1/2r + 1/2r)
  step-4 = ·-assoc
  step-5 : 1r · (1/2r + 1/2r) ≡ (1/2r · 2r) · (1/2r + 1/2r)
  step-5 = cong (λ p → p · (1/2r + 1/2r)) (sym (·-inverse-left (λ _ → pos-2r)))
  step-6 : 1/2r + 1/2r ≡ 1r · (1/2r + 1/2r)
  step-6 = sym ·-unit-left

_/2r : ℝ → ℝ
x /2r = 1/2r · x

pos-/2r-v : (x : ℝ) → 0r < x → 0r < x /2r
pos-/2r-v x pos-x = transport ·-null-left {λ p → p < 1/2r · x} (<-mult 0r x 1/2r pos-1/2r pos-x)

x/2r-less-than-x : (x : ℝ) → 0r < x → (x /2r) < x
x/2r-less-than-x x pos-x = step-3 where
  step-1 : x · 1/2r < x · 1r
  step-1 = <-mult 1/2r 1r x pos-x 1/2r-less-than-1r
  step-2 : x /2r < x · 1r
  step-2 = transport (·-comm {x} {1/2r}) {λ p → p < x · 1r} step-1
```

```
  step-3 : x /2r < x
  step-3 = transport (·-unit-right {x}) {λ p → x /2r < p} step-2

/2r-half : ∀ {x : ℝ} → x /2r + x /2r ≡ x
/2r-half {x} = tran step-1 step-2 where
  step-1 : x /2r + x /2r ≡ (1/2r + 1/2r) · x
  step-1 = sym (distr-right {1/2r} {1/2r} {x})
  step-2 : (1/2r + 1/2r) · x ≡ x
  step-2 = tran (cong (λ p → p · x) 1/2r-half) ·-unit-left

-- We prove some results about ≤ that we need for Lipschitz conditions.

_≤ᵣ_ : ℝ → ℝ → Set
a ≤ᵣ b = (a ≡ b) ∨ (a < b)

≤ᵣ-tran : ∀ x y z → x ≤ᵣ y → y ≤ᵣ z → x ≤ᵣ z
≤ᵣ-tran x .x .x (inl refl) (inl refl) = inl refl
≤ᵣ-tran x .x z (inl refl) (inr p) = inr p
≤ᵣ-tran x y .y (inr p) (inl refl) = inr p
≤ᵣ-tran x y z (inr p) (inr q) = inr (<-tran x y z p q)
≤ᵣ-plus : ∀ x y c → x ≤ᵣ y → (x + c) ≤ᵣ (y + c)
≤ᵣ-plus x .x c (inl refl) = inl refl
≤ᵣ-plus x y c (inr p) = inr (<-plus x y c p)
≤ᵣ-mult : ∀ x y c → 0r ≤ᵣ c → x ≤ᵣ y → (c · x) ≤ᵣ (c · y)
≤ᵣ-mult x .x .0r (inl refl) (inl refl) = inl refl
≤ᵣ-mult x y .0r (inl refl) (inr q) = inl (tran ·-null-right (sym ·-null-right))
≤ᵣ-mult x .x c (inr p) (inl refl) = inl refl
≤ᵣ-mult x y c (inr p) (inr q) = inr (<-mult x y c p q)
≤ᵣ-nontrivial : 0r ≤ᵣ 1r
≤ᵣ-nontrivial = inr <-nontrivial

≤ᵣ-minus : ∀ {x : ℝ} → 0r ≤ᵣ x → minus x ≤ᵣ 0r
≤ᵣ-minus (inl refl) = inl (tran (sym +-unit-left) +-inverse-left)
≤ᵣ-minus (inr p) = inr (<-minus p)

≤ᵣ-plus-both : ∀ (x X y Y : ℝ) → x ≤ᵣ X → y ≤ᵣ Y → x + y ≤ᵣ X + Y
≤ᵣ-plus-both x .x y .y (inl refl) (inl refl) = inl refl
≤ᵣ-plus-both x .x y Y (inl refl) (inr q) = inr step-3 where
  step-1 : y + x < Y + x
  step-1 = <-plus y Y x q
  step-2 : x + y < Y + x
  step-2 = transport +-comm {λ p → p < Y + x} step-1
  step-3 : x + y < x + Y
  step-3 = transport +-comm {λ p → x + y < p} step-2
≤ᵣ-plus-both x X y .y (inr p) (inl refl) = inr (<-plus x X y p)
≤ᵣ-plus-both x X y Y (inr p) (inr q) = inr (<-plus-both x X y Y p q)

≤ᵣ-plus-left : ∀ x y c → x ≤ᵣ y → (c + x) ≤ᵣ (c + y)
≤ᵣ-plus-left x y c p = step-3 where
  step-1 : x + c ≤ᵣ y + c
  step-1 = ≤ᵣ-plus x y c p
  step-2 : c + x ≤ᵣ y + c
  step-2 = transport +-comm {λ p → p ≤ᵣ y + c} step-1
  step-3 : c + x ≤ᵣ c + y
  step-3 = transport +-comm {λ p → c + x ≤ᵣ p} step-2

≤ᵣ-dichotomy : ∀ x y → (x ≤ᵣ y) ∨ (y ≤ᵣ x)
≤ᵣ-dichotomy x y with <-trichotomy-strong x y
≤ᵣ-dichotomy x y | inl x-equals-y = inl (inl x-equals-y)
≤ᵣ-dichotomy x y | inr (inl x-under-y) = inl (inr x-under-y)
≤ᵣ-dichotomy x y | inr (inr y-under-x) = inr (inr y-under-x)

lemma-lesser : ∀ (x y : ℝ) → (∀ (ε : ℝ) → 0r < ε → x ≤ᵣ y + ε) → ∀ (ε : ℝ) → 0r < ε → x < y + ε
lemma-lesser x y p ε pos-ε = step-3 where
  step-1 : x ≤ᵣ y + (ε /2r)
  step-1 = p (ε /2r) (pos-/2r-v ε pos-ε)
  step-2 : y + (ε /2r) < y + ε
  step-2 = <-plus-left (ε /2r) ε y (x/2r-less-than-x ε pos-ε)
  step-3 : x < y + ε
  step-3 with step-1
  step-3 | inl refl = step-2
  step-3 | inr p = <-tran _ _ _ p step-2

lemma-ε-of-room-plus-≤ᵣ : ∀ (x y : ℝ) → (∀ (ε : ℝ) → 0r < ε → x ≤ᵣ y + ε) → x ≤ᵣ y
lemma-ε-of-room-plus-≤ᵣ x y p with ≤ᵣ-dichotomy x y
```

```
lemma-ε-of-room-plus-≤ᵣ x y p | inl (inl x-equals-y) = inl x-equals-y
lemma-ε-of-room-plus-≤ᵣ x y p | inl (inr x-under-y) = inr x-under-y
lemma-ε-of-room-plus-≤ᵣ x y p | inr (inl y-equals-x) = inl (sym y-equals-x)
lemma-ε-of-room-plus-≤ᵣ x y p | inr (inr y-under-x) =
  inr (lemma-ε-of-room-plus x y p' x-neq-y) where
  p' : ∀ (ε : ℝ) → 0r < ε → x < y + ε
  p' = lemma-lesser x y p
  x-neq-y : x ≡ y → ⊥
  x-neq-y x-equals-y = <-asym-1 y y (transport x-equals-y {λ p → y < p} y-under-x) refl

--------------------------------------------------------------------------------


module IST.Safe.Groups where

open import IST.Safe.Base
open import IST.Safe.FiniteSets
open import IST.Safe.Naturals

record IsGroup
  (Carrier : Set)
  (identity : Carrier)
  (operation : Carrier → Carrier → Carrier)
  (inverse : Carrier → Carrier)
  : Set where
  field
    assoc : ∀ (x y z : Carrier) → operation (operation x y) z ≡ operation x (operation y z)
    unit-left : ∀ (x : Carrier) → operation identity x ≡ x
    unit-right : ∀ (x : Carrier) → operation x identity ≡ x
    inverse-left : ∀ (x : Carrier) → operation (inverse x) x ≡ identity
    inverse-right : ∀ (x : Carrier) → operation x (inverse x) ≡ identity
  power : Carrier → ℕ → Carrier
  power x zero = identity
  power x (suc n) = operation x (power x n)

record Group : Set₁ where
  field
    Carrier : Set
    identity : Carrier
    operation : Carrier → Carrier → Carrier
    inverse : Carrier → Carrier
    isGroup : IsGroup Carrier identity operation inverse
  open IsGroup isGroup public


record IsPeriodicGroup
  (Carrier : Set)
  (identity : Carrier)
  (operation : Carrier → Carrier → Carrier)
  (inverse : Carrier → Carrier)
  : Set where
  field
    isGroup : IsGroup Carrier identity operation inverse
  open IsGroup isGroup
  field
    order : Carrier → ℕ
    order-identity : ∀ g → power g (order g) ≡ identity
    order-minimal : ∀ g → ∀ n → power g (suc n) ≡ identity → order g ≤ (suc n)
    order-nonzero : ∀ g → order g ≡ 0 → ⊥

record PeriodicGroup : Set₁ where
  field
    Carrier : Set
    identity : Carrier
    operation : Carrier → Carrier → Carrier
    inverse : Carrier → Carrier
    isPeriodicGroup : IsPeriodicGroup Carrier identity operation inverse
  open IsPeriodicGroup isPeriodicGroup public
  open IsGroup isGroup public
  asGroup : Group
  asGroup =
    record { Carrier = Carrier
           ; identity = identity
           ; operation = operation
           ; inverse = inverse
```

```
            ; isGroup = isGroup
            }
  power-lemma : ∀ g → ∀ n → g ≡ identity → power g n ≡ identity
  power-lemma .(identity) zero refl = refl
  power-lemma .(identity) (suc n) refl = tran (unit-left (power identity n)) inductive-
hypothesis where
    inductive-hypothesis : power identity n ≡ identity
    inductive-hypothesis = power-lemma identity n refl

  power-lemma-contrapositive : ∀ g → ∀ n → (power g n ≡ identity → ⊥) → g ≡ identity → ⊥
  power-lemma-contrapositive g n gn-not-identity g-identity = gn-not-identity (power-lemma g n
g-identity)


record IsFiniteGroup
  (Carrier : Set)
  (identity : Carrier)
  (operation : Carrier → Carrier → Carrier)
  (inverse : Carrier → Carrier)
  : Set where
  field
    isGroup : IsGroup Carrier identity operation inverse
  open IsGroup isGroup
  field
    isFiniteSet : IsFiniteSet Carrier
    order : Carrier → ℕ
    order-identity : ∀ g → power g (order g) ≡ identity
    order-minimal : ∀ g → ∀ n → power g (suc n) ≡ identity → order g ≤ suc n
    order-nonzero : ∀ g → order g ≡ 0 → ⊥

record FiniteGroup : Set₁ where
  field
    Carrier : Set
    identity : Carrier
    operation : Carrier → Carrier → Carrier
    inverse : Carrier → Carrier
    isFiniteGroup : IsFiniteGroup Carrier identity operation inverse
  open IsFiniteGroup isFiniteGroup public
  open IsGroup isGroup public
  asGroup : Group
  asGroup =
    record { Carrier = Carrier
           ; identity = identity
           ; operation = operation
           ; inverse = inverse
           ; isGroup = isGroup
           }
  power-lemma : ∀ g → ∀ n → g ≡ identity → power g n ≡ identity
  power-lemma .(identity) zero refl = refl
  power-lemma .(identity) (suc n) refl = tran (unit-left (power identity n)) inductive-
hypothesis where
    inductive-hypothesis : power identity n ≡ identity
    inductive-hypothesis = power-lemma identity n refl

  power-lemma-contrapositive : ∀ g → ∀ n → (power g n ≡ identity → ⊥) → g ≡ identity → ⊥
  power-lemma-contrapositive g n gn-not-identity g-identity = gn-not-identity (power-lemma g n
g-identity)


record IsFiniteSubgroup
  (Source : FiniteGroup)
  (Target : Group)
  (Map : FiniteGroup.Carrier Source → Group.Carrier Target)
  : Set where
  open FiniteGroup Source public
  field
    Map-identity : Map identity ≡ Group.identity Target
    Map-operation : ∀ g h →
      Map (operation g h) ≡ Group.operation Target (Map g) (Map h)
    Map-injective : ∀ g h → Map g ≡ Map h → g ≡ h

record FiniteSubgroup (Target : Group) : Set₁ where
  field
    Source : FiniteGroup
    Map : FiniteGroup.Carrier Source → Group.Carrier Target
```

```
      isFiniteSubgroup : IsFiniteSubgroup Source Target Map
    open IsFiniteSubgroup isFiniteSubgroup public
    Map-power : ∀ g → ∀ n → Map (power g n) ≡ Group.power Target (Map g) n
    Map-power g zero = Map-identity
    Map-power g (suc n) = tran (Map-operation g gn) step-1 where
      gn : Carrier
      gn = power g n
      mgn : Group.Carrier Target
      mgn = Group.power Target (Map g) n
      inductive-hypothesis : Map gn ≡ mgn
      inductive-hypothesis = Map-power g n
      step-1 : Group.operation Target (Map g) (Map gn) ≡ Group.operation Target (Map g) mgn
      step-1 = cong (Group.operation Target (Map g)) inductive-hypothesis


  record IsPeriodicSubgroup
    (Source : PeriodicGroup)
    (Target : Group)
    (Map : PeriodicGroup.Carrier Source → Group.Carrier Target)
    : Set where
    open PeriodicGroup Source public
    field
      Map-identity : Map identity ≡ Group.identity Target
      Map-operation : ∀ g h →
        Map (operation g h) ≡ Group.operation Target (Map g) (Map h)
      Map-injective : ∀ g h → Map g ≡ Map h → g ≡ h

  record PeriodicSubgroup (Target : Group) : Set₁ where
    field
      Source : PeriodicGroup
      Map : PeriodicGroup.Carrier Source → Group.Carrier Target
      isPeriodicSubgroup : IsPeriodicSubgroup Source Target Map
    open IsPeriodicSubgroup isPeriodicSubgroup public
    Map-power : ∀ g → ∀ n → Map (power g n) ≡ Group.power Target (Map g) n
    Map-power g zero = Map-identity
    Map-power g (suc n) = tran (Map-operation g gn) step-1 where
      gn : Carrier
      gn = power g n
      mgn : Group.Carrier Target
      mgn = Group.power Target (Map g) n
      inductive-hypothesis : Map gn ≡ mgn
      inductive-hypothesis = Map-power g n
      step-1 : Group.operation Target (Map g) (Map gn) ≡ Group.operation Target (Map g) mgn
      step-1 = cong (Group.operation Target (Map g)) inductive-hypothesis

  --------------------------------------------------------------------------------


  module IST.Safe.MetricSpaces where

  open import IST.Safe.Base
  open import IST.Safe.Reals

  record IsMetricSpace
    (Carrier : Set)
    (distance : Carrier → Carrier → ℝ)
    : Set where
    field
      nonnegative : ∀ x y → distance x y < 0r → ⊥
      reflexive-1 : ∀ x y → distance x y ≡ 0r → x ≡ y
      reflexive-2 : ∀ x → distance x x ≡ 0r
      symmetry : ∀ x y → distance x y ≡ distance y x
      triangle-≤ᵣ : ∀ x y z → distance x z ≤ᵣ distance x y + distance y z
    triangle : ∀ x y z b → (distance x y + distance y z < b) → distance x z < b
    triangle x y z b p with triangle-≤ᵣ x y z
    triangle x y z b p | inl eq = transport (sym eq) {λ p → p < b} p
    triangle x y z b p | inr lt = <-tran _ _ _ lt p

  record MetricSpace : Set₁ where
    field
      Carrier : Set
      distance : Carrier → Carrier → ℝ
      isMetricSpace : IsMetricSpace Carrier distance
    open IsMetricSpace isMetricSpace public
  --------------------------------------------------------------------------------
```

```
module IST.Safe.GroupActions where

open import IST.Safe.Base
open import IST.Safe.Naturals
open import IST.Safe.Reals
open import IST.Safe.MetricSpaces
open import IST.Safe.Groups

record IsGroupAction
  (Source : Group)
  (Target : Set)
  (Map : Group.Carrier Source → Target → Target)
  : Set where
  open Group Source
  field
    action-identity : ∀ m → Map identity m ≡ m
    action-operation : ∀ g h → ∀ m → Map g (Map h m) ≡ Map (operation g h) m

record GroupAction (Source : Group) (Target : Set) : Set where
  field
    Map : Group.Carrier Source → Target → Target
    isGroupAction : IsGroupAction Source Target Map
  open IsGroupAction isGroupAction public


record IsDiscreteAction
  (Source : FiniteGroup)
  (Target : MetricSpace)
  (Map : FiniteGroup.Carrier Source →
         MetricSpace.Carrier Target → MetricSpace.Carrier Target)
  : Set where
  open FiniteGroup Source
  open MetricSpace Target
  field
    isGroupAction : IsGroupAction (FiniteGroup.asGroup Source) (MetricSpace.Carrier Target) Map
    continuity : ∀ (g : FiniteGroup.Carrier Source) →
                 ∀ (m : MetricSpace.Carrier Target) →
                 ∀ (ε : ℝ) → 0r < ε → ∃ λ (δ : ℝ) → (0r < δ) ∧ (
                 ∀ (m' : MetricSpace.Carrier Target) →
                 distance m m' < δ →
                 distance (Map g m) (Map g m') < ε)


record DiscreteAction (Source : FiniteGroup) (Target : MetricSpace) : Set where
  field
    Map : FiniteGroup.Carrier Source →
          MetricSpace.Carrier Target → MetricSpace.Carrier Target
    isDiscreteAction : IsDiscreteAction Source Target Map
  open IsDiscreteAction isDiscreteAction public
  open IsGroupAction isGroupAction public

  power-faithful : ∀ (g : FiniteGroup.Carrier Source) →
                   ∀ (m : MetricSpace.Carrier Target) →
                   ∀ (n : ℕ) → Map g m ≡ m → Map (FiniteGroup.power Source g n) m ≡ m
  power-faithful g m zero gm-equals-m = action-identity m
  power-faithful g m (suc n) gm-equals-m = tran (tran step-1 step-2) gm-equals-m where
    inductive-hypothesis : Map (FiniteGroup.power Source g n) m ≡ m
    inductive-hypothesis = power-faithful g m n gm-equals-m
    step-1 : Map (FiniteGroup.power Source g (suc n)) m ≡
             Map g (Map (FiniteGroup.power Source g n) m)
    step-1 = sym (action-operation g (FiniteGroup.power Source g n) m)
    step-2 : Map g (Map (FiniteGroup.power Source g n) m) ≡
             Map g m
    step-2 = cong (Map g) inductive-hypothesis


record IsPeriodicDiscreteAction
  (Source : PeriodicGroup)
  (Target : MetricSpace)
  (Map : PeriodicGroup.Carrier Source →
         MetricSpace.Carrier Target → MetricSpace.Carrier Target)
  : Set where
  open PeriodicGroup Source
  open MetricSpace Target
```

```
    field
      isGroupAction : IsGroupAction (PeriodicGroup.asGroup Source) (MetricSpace.Carrier Target)
  Map
      continuity : ∀ (g : PeriodicGroup.Carrier Source) →
                   ∀ (m : MetricSpace.Carrier Target) →
                   ∀ (ε : ℝ) → 0r < ε → ∃ λ (δ : ℝ) → (0r < δ) ∧ (
                   ∀ (m' : MetricSpace.Carrier Target) →
                   distance m m' < δ →
                   distance (Map g m) (Map g m') < ε)


  record PeriodicDiscreteAction (Source : PeriodicGroup) (Target : MetricSpace) : Set where
    field
      Map : PeriodicGroup.Carrier Source →
            MetricSpace.Carrier Target → MetricSpace.Carrier Target
      isPeriodicDiscreteAction : IsPeriodicDiscreteAction Source Target Map
    open IsPeriodicDiscreteAction isPeriodicDiscreteAction public
    open IsGroupAction isGroupAction public

    power-faithful : ∀ (g : PeriodicGroup.Carrier Source) →
                     ∀ (m : MetricSpace.Carrier Target) →
                     ∀ (n : ℕ) → Map g m ≡ m → Map (PeriodicGroup.power Source g n) m ≡ m
    power-faithful g m zero gm-equals-m = action-identity m
    power-faithful g m (suc n) gm-equals-m = tran (tran step-1 step-2) gm-equals-m where
      inductive-hypothesis : Map (PeriodicGroup.power Source g n) m ≡ m
      inductive-hypothesis = power-faithful g m n gm-equals-m
      step-1 : Map (PeriodicGroup.power Source g (suc n)) m ≡
               Map g (Map (PeriodicGroup.power Source g n) m)
      step-1 = sym (action-operation g (PeriodicGroup.power Source g n) m)
      step-2 : Map g (Map (PeriodicGroup.power Source g n) m) ≡
               Map g m
      step-2 = cong (Map g) inductive-hypothesis

  --------------------------------------------------------------------------------


  module IST.Safe.NewmansTheorem where

  open import IST.Safe.Base
  open import IST.Safe.Naturals
  open import IST.Safe.Reals
  open import IST.Safe.MetricSpaces
  open import IST.Safe.Groups
  open import IST.Safe.GroupActions


  -- Formally proving Newman's theorem lies outside the scope of our work, and so
  -- we do not give a definition of compact metric manifolds. Instead, we work with
  -- Newman spaces: metric spaces that satisfy Corollary 2.3.6. By Newman's theorem
  -- (Theorem 2.3.5.) all compact metric manifolds form Newman spaces.

  record IsNewmanSpace
    (M : MetricSpace)
    (ν : ℝ)
    : Set₁ where
    open MetricSpace M
    field
      isPositive : 0r < ν
      isNewmanConstant :
        ∀ (G : FiniteGroup) →
        ∀ (g : FiniteGroup.Carrier G) → (g ≡ FiniteGroup.identity G → ⊥) →

        ∀ (A : DiscreteAction G M) →

        (∀ (x : FiniteGroup.Carrier G) → (x ≡ FiniteGroup.identity G → ⊥) →
        ∃ λ (m : Carrier) → DiscreteAction.Map A x m ≡ m → ⊥) →

        ∃ λ (n : ℕ) → ∃ λ (m : Carrier) → (n ≤ FiniteGroup.order G g) ∧
        (ν < distance m (DiscreteAction.Map A (FiniteGroup.power G g n) m))

  record NewmanSpace : Set₁ where
    field
      asMetricSpace : MetricSpace
      inhabitant : MetricSpace.Carrier asMetricSpace
      newman-constant : ℝ
```

```
    isNewmanSpace : IsNewmanSpace asMetricSpace newman-constant
  open MetricSpace asMetricSpace public
  open IsNewmanSpace isNewmanSpace public

--------------------------------------------------------------------------------


module IST.Safe.Validation where

-- T. Chow on Hirsch-style criticism of mechanized proofs:
-- "As you know, one thing that a skeptic can say even when shown a formal
-- proof is, Yes, you've produced a formal proof of *something*, but what
-- you've proved isn't the statement that we know [..]"

-- To avoid Hirsch-style criticism, we give some basic examples to convince
-- the reader that our notion of group, periodic group, finite group, metric
-- space corresponds to the usual notions.

open import Agda.Primitive
open import IST.Safe.Base
open import IST.Safe.Naturals
open import IST.Safe.FiniteSets
open import IST.Safe.Reals
open import IST.Safe.Groups
open import IST.Safe.GroupActions
open import IST.Safe.MetricSpaces
open import IST.Safe.NewmansTheorem

-- ℤ/2ℤ forms a (finite, a fortiori periodic) group.

data ℤ₂ : Set where
  0₂ : ℤ₂
  1₂ : ℤ₂

infixl 10 _+₂_
_+₂_ : ℤ₂ → ℤ₂ → ℤ₂
0₂ +₂ y = y
1₂ +₂ 0₂ = 1₂
1₂ +₂ 1₂ = 0₂

+₂-assoc : ∀ (x y z : ℤ₂) → x +₂ y +₂ z ≡ x +₂ (y +₂ z)
+₂-assoc 0₂ y z = refl
+₂-assoc 1₂ 0₂ z = refl
+₂-assoc 1₂ 1₂ 0₂ = refl
+₂-assoc 1₂ 1₂ 1₂ = refl

+₂-unit-right : ∀ (x : ℤ₂) → x +₂ 0₂ ≡ x
+₂-unit-right 0₂ = refl
+₂-unit-right 1₂ = refl

+₂-inverse : ∀ (x : ℤ₂) → x +₂ x ≡ 0₂
+₂-inverse 0₂ = refl
+₂-inverse 1₂ = refl

ℤ/2ℤ : Group
ℤ/2ℤ = record
         { Carrier = ℤ₂
         ; identity = 0₂
         ; operation = _+₂_
         ; inverse = λ x → x
         ; isGroup = record
                       { assoc = +₂-assoc
                       ; unit-left = λ _ → refl
                       ; unit-right = +₂-unit-right
                       ; inverse-left = +₂-inverse
                       ; inverse-right = +₂-inverse
                       }
         }

order₂ : ℤ₂ → ℕ
order₂ 0₂ = suc zero
order₂ 1₂ = suc (suc zero)
```

```
order₂-identity : (g : ℤ₂) → IsGroup.power (Group.isGroup ℤ/2ℤ) g (order₂ g) ≡ 0₂
order₂-identity 0₂ = refl
order₂-identity 1₂ = refl

order₂-nonzero : (g : ℤ₂) → order₂ g ≡ 0 → ⊥
order₂-nonzero 0₂ ()
order₂-nonzero 1₂ ()

order₂-minimal : (g : ℤ₂) → (n : ℕ) → IsGroup.power (Group.isGroup ℤ/2ℤ) g (suc n) ≡ 0₂ → order₂
g ≤ suc n
order₂-minimal 0₂ n p = ≤-suc ≤-zero
order₂-minimal 1₂ (suc zero) refl = ≤-suc (≤-suc ≤-zero)
order₂-minimal 1₂ (suc (suc n)) p = ≤-suc (≤-suc ≤-zero)

ℤ/2ℤ' : PeriodicGroup
ℤ/2ℤ' = record
          { Carrier = ℤ₂
          ; identity = 0₂
          ; operation = _+₂_
          ; inverse = λ x → x
          ; isPeriodicGroup = record
                                { isGroup =  Group.isGroup ℤ/2ℤ
                                ; order = order₂
                                ; order-identity = order₂-identity
                                ; order-minimal = order₂-minimal
                                ; order-nonzero = order₂-nonzero
                                }
          }

-- the order is determined by the definition

order₂-unique : ∀ (p : IsPeriodicGroup ℤ₂ 0₂ _+₂_ (λ x → x)) →
                (IsPeriodicGroup.order p 0₂ ≡ 1) ∧ (IsPeriodicGroup.order p 1₂ ≡ 2)
order₂-unique p = ord-0₂-equals-1 , ord-1₂-equals-2 where
  open IsPeriodicGroup p
  ord-0₂-equals-1 : order 0₂ ≡ 1
  ord-0₂-equals-1 = lemma (order 0₂) (order-minimal 0₂ zero refl) (order-nonzero 0₂) where
    lemma : ∀ x → x ≤ 1 → (x ≡ 0 → ⊥) → x ≡ 1
    lemma zero p q = absurd (q refl)
    lemma (suc .0) (≤-suc ≤-zero) q = refl
  ord-1₂-under-2 : order 1₂ ≤ 2
  ord-1₂-under-2 = order-minimal 1₂ (suc zero) refl
  ord-1₂-neq-1 : order 1₂ ≡ 1 → ⊥
  ord-1₂-neq-1 assumption = absurd (0₂-neq-1₂ 0₂-equals-1₂) where
    step-1 : IsGroup.power isGroup 1₂ (order 1₂) ≡ 0₂
    step-1 = order-identity 1₂
    step-2 : IsGroup.power isGroup 1₂ 1 ≡ 0₂
    step-2 = transport assumption {λ p → IsGroup.power isGroup 1₂ p ≡ 0₂} step-1
    step-3 : IsGroup.power isGroup 1₂ 1 ≡ 1₂
    step-3 = refl
    0₂-neq-1₂ : 0₂ ≡ 1₂ → ⊥
    0₂-neq-1₂ ()
    0₂-equals-1₂ : 0₂ ≡ 1₂
    0₂-equals-1₂ = tran (sym step-2) step-3
  ord-1₂-equals-2 : order 1₂ ≡ 2
  ord-1₂-equals-2 = lemma (order 1₂) ord-1₂-under-2 (order-nonzero 1₂) ord-1₂-neq-1 where
    lemma : ∀ x → x ≤ 2 → (x ≡ 0 → ⊥) → (x ≡ 1 → ⊥) → x ≡ 2
    lemma .0 ≤-zero q r = absurd (q refl)
    lemma .1 (≤-suc ≤-zero) q r = absurd (r refl)
    lemma .2 (≤-suc (≤-suc ≤-zero)) q r = refl


-- ℤ₂ forms a finite group

list₂ : List ℤ₂
list₂ = 0₂ ∷ (1₂ ∷ [])

has-all-elements₂ : ∀ (x : ℤ₂) → x ∈ list₂
has-all-elements₂ 0₂ = ∈-head
has-all-elements₂ 1₂ = ∈-tail ∈-head
```

```
finite₂ : IsFiniteSet ℤ₂
finite₂ = record { list-of-elements = list₂ ; has-all-elements = has-all-elements₂ }

ℤ/2ℤ'' : FiniteGroup
ℤ/2ℤ'' = record
           { Carrier = ℤ₂
           ; identity = 0₂
           ; operation = _+₂_
           ; inverse = λ x → x
           ; isFiniteGroup = record
                                 { isGroup = Group.isGroup ℤ/2ℤ
                                 ; isFiniteSet = finite₂
                                 ; order = order₂
                                 ; order-identity = order₂-identity
                                 ; order-minimal = order₂-minimal
                                 ; order-nonzero = order₂-nonzero
                                 }
           }

-- The set of natural numbers is not finite.

infinite-ℕ : IsFiniteSet ℕ → ⊥
infinite-ℕ finite-ℕ = ≤-not-suc M contradiction where
  open IsFiniteSet finite-ℕ renaming (list-of-elements to list; has-all-elements to all)
  max : ∀ (x y : ℕ) → ∃ λ M → (x ≤ M) ∧ (y ≤ M)
  max zero y = y , ≤-zero , ≤-refl y
  max (suc x) zero = suc x , ≤-refl (suc x) , ≤-zero
  max (suc x) (suc y) with max x y
  max (suc x) (suc y) | M , x≤M , y≤M = suc M , ≤-suc x≤M , ≤-suc y≤M
  maximum : ∀ (nats : List ℕ) → ∃ λ M → ∀ z → z ∈ nats → z ≤ M
  maximum [] = zero , (λ x ())
  maximum (x ∷ xs) with maximum xs
  maximum (x ∷ xs) | M , M-dominates-xs with max x M
  maximum (x ∷ xs) | M , M-dominates-xs | M' , x≤M' , M≤M' = M' , M'-dominates-xs where
    M'-dominates-xs : ∀ z → z ∈ (x ∷ xs) → z ≤ M'
    M'-dominates-xs z ∈-head = x≤M'
    M'-dominates-xs z (∈-tail p) = ≤-tran z M M' (M-dominates-xs z p) M≤M'
  M : ℕ
  M = proj₁ (maximum list)
  M-dominates-list : ∀ (x : ℕ) → x ∈ list → x ≤ M
  M-dominates-list = proj₂ (maximum list)
  M-largest : ∀ (x : ℕ) → x ≤ M
  M-largest x = M-dominates-list x (all x)
  contradiction : suc M ≤ M
  contradiction = M-largest (suc M)

-- Metric spaces exist, in particular the discrete metric is a metric.

discrete : ℤ₂ → ℤ₂ → ℝ
discrete 0₂ 0₂ = 0r
discrete 0₂ 1₂ = 1r
discrete 1₂ 0₂ = 1r
discrete 1₂ 1₂ = 0r

discrete-nonnegative : ∀ (x y : ℤ₂) → discrete x y < 0r → ⊥
discrete-nonnegative 0₂ 0₂ p = <-asym-1 0r 0r p refl
discrete-nonnegative 0₂ 1₂ p = <-asym-2 0r 1r <-nontrivial p
discrete-nonnegative 1₂ 0₂ p = <-asym-2 0r 1r <-nontrivial p
discrete-nonnegative 1₂ 1₂ p = <-asym-1 0r 0r p refl

discrete-reflexive-1 : ∀ (x y : ℤ₂) → discrete x y ≡ 0r → x ≡ y
discrete-reflexive-1 0₂ 0₂ refl = refl
discrete-reflexive-1 0₂ 1₂ p = absurd (<-asym-1 0r 1r <-nontrivial (sym p))
discrete-reflexive-1 1₂ 0₂ p = absurd (<-asym-1 0r 1r <-nontrivial (sym p))
discrete-reflexive-1 1₂ 1₂ refl = refl

discrete-reflexive-2 : ∀ (x : ℤ₂) → discrete x x ≡ 0r
discrete-reflexive-2 0₂ = refl
discrete-reflexive-2 1₂ = refl

discrete-symmetry : ∀ (x y : ℤ₂) → discrete x y ≡ discrete y x
```

```
discrete-symmetry 0₂ 0₂ = refl
discrete-symmetry 0₂ 1₂ = refl
discrete-symmetry 1₂ 0₂ = refl
discrete-symmetry 1₂ 1₂ = refl

discrete-triangle : ∀ (x y z : ℤ₂) → discrete x z ≤ᵣ discrete x y + discrete y z
discrete-triangle 0₂ 0₂ 0₂ = inl (sym +-unit-left)
discrete-triangle 0₂ 0₂ 1₂ = inl (sym +-unit-left)
discrete-triangle 0₂ 1₂ 0₂ = inr pos-2r
discrete-triangle 0₂ 1₂ 1₂ = inl (sym +-unit-right)
discrete-triangle 1₂ 0₂ 0₂ = inl (sym +-unit-right)
discrete-triangle 1₂ 0₂ 1₂ = inr pos-2r
discrete-triangle 1₂ 1₂ 0₂ = inl (sym +-unit-left)
discrete-triangle 1₂ 1₂ 1₂ = inl (sym +-unit-left)

ℤ₂-metric : MetricSpace
ℤ₂-metric =
  record { Carrier = ℤ₂
         ; distance = discrete
         ; isMetricSpace = record
             { nonnegative = discrete-nonnegative
             ; reflexive-1 = discrete-reflexive-1
             ; reflexive-2 = discrete-reflexive-2
             ; symmetry = discrete-symmetry
             ; triangle-≤ᵣ = discrete-triangle
             }
         }

-- Faithful, K-Lipschitz actions exist.

act : ℤ₂ → ℤ₂ → ℤ₂
act x y = x +₂ y

act-identity : ∀ m → act 0₂ m ≡ m
act-identity = Group.unit-left ℤ/2ℤ

act-operation : ∀ (g h : ℤ₂) → ∀ m → act g (act h m) ≡ act (g +₂ h) m
act-operation g h m = sym (+₂-assoc g h m)

action₂ : GroupAction ℤ/2ℤ ℤ₂
action₂ = record
  { Map = act
  ; isGroupAction = record
    { action-identity = act-identity
    ; action-operation = act-operation
    }
  }

act-faithful : ∀ (g : ℤ₂) → (g ≡ 0₂ → ⊥) → ∃ λ (m : ℤ₂) → act g m ≡ m → ⊥
act-faithful 0₂ p = absurd (p refl)
act-faithful 1₂ p = 1₂ , lemma 1₂ where
  lemma : ∀ x → act 1₂ x ≡ x → ⊥
  lemma 0₂ ()
  lemma 1₂ ()

K : ℝ
K = 1r

act-lipschitz : ∀ (g : ℤ₂) → ∀ (x y : ℤ₂) → discrete (act g x) (act g y) ≤ᵣ (K · discrete x y)
act-lipschitz 0₂ 0₂ 0₂ = transport (sym (·-null-left {K})) {λ p → 0r ≤ᵣ p} (inl refl)
act-lipschitz 0₂ 0₂ 1₂ = transport (sym (·-unit-right {K})) {λ p → 1r ≤ᵣ p} (inl refl)
act-lipschitz 0₂ 1₂ 0₂ = transport (sym (·-unit-right {K})) {λ p → 1r ≤ᵣ p} (inl refl)
act-lipschitz 0₂ 1₂ 1₂ = transport (sym (·-null-left {K})) {λ p → 0r ≤ᵣ p} (inl refl)
act-lipschitz 1₂ 0₂ 0₂ = transport (sym (·-null-left {K})) {λ p → 0r ≤ᵣ p} (inl refl)
act-lipschitz 1₂ 0₂ 1₂ = transport (sym (·-unit-right {K})) {λ p → 1r ≤ᵣ p} (inl refl)
act-lipschitz 1₂ 1₂ 0₂ = transport (sym (·-unit-right {K})) {λ p → 1r ≤ᵣ p} (inl refl)
act-lipschitz 1₂ 1₂ 1₂ = transport (sym (·-null-left {K})) {λ p → 0r ≤ᵣ p} (inl refl)

-- Newman spaces exist.

nonzero-lemma : ∀ n → (n ≡ 0 → ⊥) → 1 ≤ n
nonzero-lemma zero p = absurd (p refl)
```

```
nonzero-lemma (suc n) p = ≤-suc ≤-zero

alt-lemma-1 : ∀ g → (g ≡ 0₂ → ⊥) → g ≡ 1₂
alt-lemma-1 0₂ p = absurd (p refl)
alt-lemma-1 1₂ p = refl

alt-lemma-0 : ∀ g → (g ≡ 1₂ → ⊥) → g ≡ 0₂
alt-lemma-0 0₂ p = refl
alt-lemma-0 1₂ p = absurd (p refl)

ℤ₂-newman : NewmanSpace
ℤ₂-newman = record
              { asMetricSpace = ℤ₂-metric
              ; inhabitant = 0₂
              ; newman-constant = 1/2r
              ; isNewmanSpace = record
                { isPositive = pos-1/2r
                ; isNewmanConstant = newman
                }
              } where
  newman : (G : FiniteGroup) (g : FiniteGroup.Carrier G) → (g ≡ FiniteGroup.identity G → ⊥) →
           (A : DiscreteAction G ℤ₂-metric) →
           ( ∀ x → (x ≡ FiniteGroup.identity G → ⊥) → ∃ λ m → DiscreteAction.Map A x m ≡ m → ⊥)
→
           ∃ λ n → ∃ λ m → (n ≤ FiniteGroup.order G g) ∧
           (1/2r < MetricSpace.distance ℤ₂-metric m (DiscreteAction.Map A (FiniteGroup.power G g
n) m))
  newman G g p A nontriv-A  with nontriv-A g p
  newman G g p A nontriv-A | 0₂ , q = 1 , 0₂ , nonzero-lemma _ step-1 , step-3 where
    step-1 : IsFiniteGroup.order (FiniteGroup.isFiniteGroup G) g ≡ 0 → ⊥
    step-1 = FiniteGroup.order-nonzero G g
    m : ℤ₂
    m = DiscreteAction.Map A (FiniteGroup.operation G g (FiniteGroup.identity G)) 0₂
    m' : ℤ₂
    m' = DiscreteAction.Map A g 0₂
    m-equals-m' : m ≡ m'
    m-equals-m' = cong (λ z → DiscreteAction.Map A z 0₂) (FiniteGroup.unit-right G g)
    m'-equals-1₂ : m' ≡ 1₂
    m'-equals-1₂ = alt-lemma-1 m' q
    1₂-equals-m : 1₂ ≡ m
    1₂-equals-m = sym (tran m-equals-m' m'-equals-1₂)
    step-2 : discrete 0₂ m ≡ 1r
    step-2 = transport 1₂-equals-m {λ z → discrete 0₂ z ≡ 1r} refl
    step-3 : 1/2r < discrete 0₂ m
    step-3 = transport (sym step-2) {λ z → 1/2r < z} 1/2r-less-than-1r
  newman G g p A nontriv-A | 1₂ , q = 1 , 1₂ , nonzero-lemma _ step-1 , step-3 where
    step-1 : IsFiniteGroup.order (FiniteGroup.isFiniteGroup G) g ≡ 0 → ⊥
    step-1 = FiniteGroup.order-nonzero G g
    m : ℤ₂
    m = DiscreteAction.Map A (FiniteGroup.operation G g (FiniteGroup.identity G)) 1₂
    m' : ℤ₂
    m' = DiscreteAction.Map A g 1₂
    m-equals-m' : m ≡ m'
    m-equals-m' = cong (λ z → DiscreteAction.Map A z 1₂) (FiniteGroup.unit-right G g)
    m'-equals-0₂ : m' ≡ 0₂
    m'-equals-0₂ = alt-lemma-0 m' q
    0₂-equals-m : 0₂ ≡ m
    0₂-equals-m = sym (tran m-equals-m' m'-equals-0₂)
    step-2 : discrete 1₂ m ≡ 1r
    step-2 = transport 0₂-equals-m {λ z → discrete 1₂ z ≡ 1r} refl
    step-3 : 1/2r < discrete 1₂ m
    step-3 = transport (sym step-2) {λ z → 1/2r < z} 1/2r-less-than-1r

--------------------------------------------------------------------------------


{-# OPTIONS --omega-in-omega #-}

module IST.Base where

open import Agda.Primitive
open import IST.Safe.Base public
```

```
-- We start by defining the sort of the external sets.
-- Internal sets belong to the first segment of the universe hierarchy,
-- while external sets belong to the second segment:
-- Set 0 : Set 1 : Set 2 : ...  Set ω : Set (ω + 1) : ...
-- \_________________________/  \________________________/
--      internal sets                external sets
-- Alas, Agda does not support higher segments of the hierarchy yet,
-- so we work under --omega-in-omega. Everything here should be typable
-- in the full hierarchy, however, by replacing some occurrences of
-- Set ω with Set (ω + 1).

ESet : Setω
ESet = Setω

ESet₁ : Setω
ESet₁ = Setω

-- We postulate a predicate st(-) asserting that its argument is standard.
-- Note that the value of st(-) lives in the external hierarchy.
-- This ensures that the type (I → Set ℓ) ranges over internal predicates
-- only, whenever ℓ < ω.

-- By declaring ST as a private data type, we ensure the following:
-- 1. st(x) is treated as a contractible type for all x.
-- 2. Outside of this module, the only way to produce a value of st(-)
--    is by using the rules/axioms presented here.

private
  data ST {ℓ : Level} {S : Set ℓ} (x : S) : ESet where
    trust-me-its-standard : ST x

st : {ℓ : Level} → {S : Set ℓ} → S → ESet
st = ST

-- A Safe module does not have access to any extended features (st predicates,
-- IST axioms, Setω), so a top-level definition `t : T` in a Safe module
-- corresponds to a derivation `⊢ˢ t : T` in extended type theory.
-- By the admissibility of the St-Con rule, we can mark any such definition
-- standard. This is accomplished by opening SafeImportTools, and using
-- the provided constructor.

module SafeImportTools where
  declared-in-safe-module : {ℓ : Level} {S : Set ℓ} (x : S) → st x
  declared-in-safe-module _ = trust-me-its-standard

-- The internal hierarchy consists only of standard universes. This follows
-- from the admissibility of the St-Con rule.

st-Set : {ℓ : Level} → st (Set ℓ)
st-Set = trust-me-its-standard

-- FUNCTION TYPES --

-- We declare that the type former ∀ (and by extension →) preserve standardness.
-- This is an easy consequence of the Transfer rules.

st-→ : {ℓ₁ ℓ₂ : Level} → (A : Set ℓ₁) → st A → (B : Set ℓ₂) → st B → st (A → B)
st-→ A st-A B st-B = trust-me-its-standard

st-∀ : {ℓ₁ ℓ₂ : Level} → (A : Set ℓ₁) → st A → (B : A → Set ℓ₂) → st B → st (∀ a → B a)
st-∀ A st-A B st-B = trust-me-its-standard

-- Function application preserves standardness, i.e. if f and x are standard,
-- then so is f(x). Notice that this principle occurs as a theorem in Nelson's
-- Internal Set Theory, and follows from St-Fun for our extended type theory.
-- We add variations for dependent and simple function types, with or without
-- hidden arguments.

st-fun-d : {ℓ₁ ℓ₂ : Level} → (A : Set ℓ₁) → (B : A → Set ℓ₂) →
           (f : (x : A) → B x) → (x : A) →
           st f → st x → st (f x)
st-fun-d A B f x st-f st-x = trust-me-its-standard

st-fun-hd : {ℓ₁ ℓ₂ : Level} → (A : Set ℓ₁) → (B : A → Set ℓ₂) →
            (f : {x : A} → B x) → (x : A) →
```

```
            st (λ x → f {x}) → st x → st (f {x})
st-fun-hd A B f x st-f st-x = st-fun-d A B (λ x → f {x}) x st-f st-x

st-fun : {ℓ₁ ℓ₂ : Level} → (A : Set ℓ₁) → (B : Set ℓ₂) →
         (f : A → B) → (x : A) →
         st f → st x → st (f x)
st-fun A B f x st-f st-x = st-fun-d A (λ _ → B) f x st-f st-x

st-fun-h : {ℓ₁ ℓ₂ : Level} → (A : Set ℓ₁) → (B : Set ℓ₂) →
           (f : {a : A} → B) → (x : A) →
           st (λ x → f {x}) → st x → st (f {x})
st-fun-h A B f x st-f st-x = st-fun A B (λ x → f {x}) x st-f st-x

-- That leaves function abstraction.
-- It would be convenient to have the following converse:
--   st-λ : {ℓ₁ ℓ₂ : Level} → (A : Set ℓ₁) → st A → (B : A → Set ℓ₂) → (∀ a → st a → st (B a)) →
st B
--   st-λ A st-A B st-Ba = trust-me-its-standard
-- Alas, this principle does not hold. Consider e.g.
-- the function f : ℕ → {0,1} with f(n)=0 ↔ n=ω, which is not
-- standard, but takes standard values everywhere.

-- So how do we prove Set-types standard? In IST, we do not
-- have to deal with this problem, since we normally encode
-- functions as their graphs (sets of ordered pairs), and IST
-- already provides rules for the standardness of sets.

-- In Agda, functions do not coincide with sets of ordered pairs,
-- and we need to ensure that all MLTT-definable functions are
-- indeed standard, even if we define them in terms of standard
-- objects constructed by Standardization, i.e. necessarily
-- outside of a Safe module. . To accomplish this, we can make the following observations:
-- 1. All combinatorial (closed) λ-terms are constructible in the Safe fragment, and hence
standard.
-- 2. The eliminators of all data types available in the Safe fragment are themselves standard.
-- 3. Applying a standard value to a standard function yields a standard result.
-- These rules exhaust all possible ways of defining functions in MLTT.

-- E.g. to prove that (λi. _=_ (f i) (g i)) is standard, we can
-- argue as follows:
-- 1. (\a.\b.\c. a b c) is a purely combinatorial λ-term, so standard.
-- 2. (\b.\c _=_ b c) is standard when both _=_ and (\a.\b.\c. a b c) are standard.
-- 3. (\c _=_ (f i) c) is standard when both (f i) and (\b.\c _=_ b c) are standard.
-- 4. (_=_ (f i) (g i)) is standard when both (g i) and (\c. _=_ (f i) (g i)) are standard.
-- So we'd conclude that the inhabitant (λi. _=_ (f i) (g i))
-- of the type Set is standard as long as (f i) and (g i) are.

-- We face one problem: the difficulty of encoding the
-- standardness of combinatorial λ-terms in Agda. To simplify
-- our life, we pre-declare instances that we actually use
-- during the present development.

st-abs-5 : (I : Set) → st (abs-5 I)
st-abs-5 I = trust-me-its-standard

st-abs-4 : st abs-4
st-abs-4 = trust-me-its-standard

st-abs-K : {ℓ₁ ℓ₂ : Level} (A : Set ℓ₁) (B : Set ℓ₂) → st (abs-K A B)
st-abs-K A B = trust-me-its-standard

st-abs-K-h : {ℓ₁ ℓ₂ : Level} (A : Set ℓ₁) (B : Set ℓ₂) → st (abs-K-h A B)
st-abs-K-h A B = trust-me-its-standard


-- TRIVIAL DATA TYPES --

absurd* : {ℓ : Level} → ⊥ → ∀ {A : ESet} → A
absurd* ()

st-⊥ : st ⊥
st-⊥ = trust-me-its-standard

st-T : st T
st-T = trust-me-its-standard
```

```
st-tt : st tt
st-tt = trust-me-its-standard


-- EXISTENTIAL QUANTIFICATION --

-- Now we deal with existential quantifiers. Alas, unlike the ∀
-- case, Agda does not provide a builtin for this, so we need to
-- declare two variants, ∃ (for the internal hierarchy) and ∃*
-- (for the external hierarchy).

st-∃ : ∀ {ℓ₁ ℓ₂ : Level} → st (λ {A : Set ℓ₁} → ∃ {ℓ₁} {ℓ₂} {A})
st-∃ = trust-me-its-standard

st-∃-full : ∀ {ℓ₁ ℓ₂ : Level} → {A : Set ℓ₁} → st (∃ {ℓ₁} {ℓ₂} {A})
st-∃-full = trust-me-its-standard

st-∃-_,_ : ∀ {ℓ₁ ℓ₂ : Level} → st (λ {A : Set ℓ₁} → λ (B : A → Set ℓ₂) → ∃._,_ {ℓ₁} {ℓ₂} {A} {B})
st-∃-_,_ = trust-me-its-standard

st-∃-_,_-full : ∀ {ℓ₁ ℓ₂ : Level} → {A : Set ℓ₁} → {B : A → Set ℓ₂} → st (∃._,_ {ℓ₁} {ℓ₂} {A} {B})
st-∃-_,_-full = trust-me-its-standard

st-∃-proj₁ : ∀ {ℓ₁ ℓ₂ : Level} → st (λ {A : Set ℓ₁} → λ (B : A → Set ℓ₂) → ∃.proj₁ {ℓ₁} {ℓ₂} {A}
{B})
st-∃-proj₁ = trust-me-its-standard

st-∃-proj₁-full : ∀ {ℓ₁ ℓ₂ : Level} → {A : Set ℓ₁} → {B : A → Set ℓ₂} → st (∃.proj₁ {ℓ₁} {ℓ₂} {A}
{B})
st-∃-proj₁-full = trust-me-its-standard

st-∃-proj₂ : ∀ {ℓ₁ ℓ₂ : Level} → st (λ {A : Set ℓ₁} → λ (B : A → Set ℓ₂) → ∃.proj₂ {ℓ₁} {ℓ₂} {A}
{B})
st-∃-proj₂ = trust-me-its-standard

st-∃-proj₂-full : ∀ {ℓ₁ ℓ₂ : Level} → {A : Set ℓ₁} → {B : A → Set ℓ₂} → st (∃.proj₂ {ℓ₁} {ℓ₂} {A}
{B})
st-∃-proj₂-full = trust-me-its-standard

st-∧ : ∀ {ℓ₁ ℓ₂ : Level} → st (_∧_ {ℓ₁} {ℓ₂})
st-∧ = trust-me-its-standard

record ∃* {ℓ : Level} {A : Set ℓ} (B : A → ESet) : ESet where
  constructor _,_
  field
    proj₁ : A
    proj₂ : B proj₁
open ∃* public

record _*∧*_ (A B : ESet) : ESet where
  constructor _,_
  field
    proj₁ : A
    proj₂ : B
open _*∧*_ public


-- LISTS / FINITE SETS --

st-List : {ℓ : Level} → st (List {ℓ})
st-List = trust-me-its-standard

st-[] : {ℓ : Level} → st (λ {A : Set ℓ} → [] {ℓ} {A})
st-[] = trust-me-its-standard

st-∷ : {ℓ : Level} → st (λ {A : Set ℓ} → _∷_ {ℓ} {A})
st-∷ = trust-me-its-standard


-- DISJUNCTION --

-- We could encode the disjunction A ∨ B using ¬ A → B, or
```

```
-- in a more constructive spirit as ∃n:ℕ. (n = 0 → A) ∧ (n ≠ 0) → B,
-- but we find it more legible to use the inductive definition, along
-- with a strong elimination principle.

st-V : {ℓ₁ ℓ₂ : Level} → st (_V_ {ℓ₁} {ℓ₂})
st-V = trust-me-its-standard

st-inl : {ℓ₁ ℓ₂ : Level} → st (λ {A : Set ℓ₁} → λ {B : Set ℓ₂} → inl {ℓ₁} {ℓ₂} {A} {B})
st-inl = trust-me-its-standard

st-inr : {ℓ₁ ℓ₂ : Level} → st (λ {A : Set ℓ₁} → λ {B : Set ℓ₂} → inr {ℓ₁} {ℓ₂} {A} {B})
st-inr = trust-me-its-standard

by-cases* : {ℓ₁ ℓ₂ : Level} {A : Set ℓ₁} {B : Set ℓ₂} →
            (P : ESet) → (A → P) → (B → P) → A V B → P
by-cases* P A-implies-P B-implies-P (inl a) = A-implies-P a
by-cases* P A-implies-P B-implies-P (inr b) = B-implies-P b


-- EQUALITY --

-- We introduced equality only for the internal hierarchy, at
-- least for now. This satisfies the usual principles.
-- We declare a variant of transport (equality preserves all
-- properties) that works for external predicates. Note that
-- this is a logical axiom in IST, which makes it invisible.
-- Technically, one should have x = y → st(x) → st(y) as an
-- axiom even there, we fix this omission in our version
-- of the Nelson translation.

transport* : {ℓ : Level} {A : Set ℓ} {x y : A} → x ≡ y → ∀ {φ : A → Setω} → φ x → φ y
transport* refl z = z

st-≡ : {ℓ : Level} → st (λ {A : Set ℓ} → _≡_ {ℓ} {A})
st-≡ = trust-me-its-standard

st-≡-full : {ℓ : Level} → {A : Set ℓ} → st (_≡_ {ℓ} {A})
st-≡-full = trust-me-its-standard

st-refl : {ℓ : Level} → st (λ {A : Set ℓ} → λ {x : A} → refl {ℓ} {A} {x})
st-refl = trust-me-its-standard

st-≡-ind : {ℓ₁ ℓ₂ : Level} → st (λ {A : Set ℓ₁} → λ {x : A} → ≡-ind {ℓ₁} {ℓ₂} {A} {x})
st-≡-ind = trust-me-its-standard


-- AXIOM: TRANSFER --

-- TransferPred implements the Transfer rules Dfun and Dsum.
-- This has the advantage that it does no branching. We do not
-- rely on Transfer for non-prenex formulae in our development,
-- so this suffices.

-- Notice that TransferPred does not satisfy strict positivity.
-- We do not export an elimination rule, so we cannot use it in
-- a dangerous/inconsistent way. If the need for an eliminator
-- ever arises, we can make it strictly positive by indexing
-- over the number of free variables.

data internal {ℓ : Level} (φ : Set ℓ) : ESet where
  fromInternal : φ → internal φ

toInternal : {ℓ : Level} → (φ : Set ℓ) → internal φ → φ
toInternal φ (fromInternal x) = x

data TransferPred : ESet where
  ∀' : (A : Set) → ((φ : A) → TransferPred) → TransferPred
  ∃' : (E : Set) → ((φ : E) → TransferPred) → TransferPred
  int' : (φ : Set) → TransferPred

toTransferI : TransferPred → Set
toTransferI (∀' A φ) = ∀ (a : A) → toTransferI (φ a)
toTransferI (∃' E φ) = ∃ λ (e : E) → toTransferI (φ e)
toTransferI (int' φ) = φ
```

```
toTransferE : TransferPred → ESet
toTransferE (∀' A φ) = ∀ (a : A) → st a → toTransferE (φ a)
toTransferE (∃' E φ) = ∃* λ (e : E) → st e *∧* toTransferE (φ e)
toTransferE (int' φ) = internal φ

std-params : TransferPred → ESet
std-params (∀' A φ) = st A *∧* ∀ (a : A) → st a → std-params (φ a)
std-params (∃' E φ) = st E *∧* ∀ (e : E) → st e → std-params (φ e)
std-params (int' φ) = st φ

postulate
  ax-Transfer-IE : (φ : TransferPred) → toTransferI φ → std-params φ → toTransferE φ
  ax-Transfer-EI : (φ : TransferPred) → toTransferE φ → std-params φ → toTransferI φ


-- AXIOM: Standardization --

postulate
  ⟦_⟧ : ∀ {ℓ} → {A : Set ℓ} → (A → ESet) → A → Set ℓ
  ax-Standard-1 : ∀ {ℓ} → {A : Set ℓ} → (φ : A → ESet) → st ⟦ φ ⟧
  ax-Standard-2 : ∀ {ℓ} → {A : Set ℓ} → (φ : A → ESet) →
    (∀ x → st x → ⟦ φ ⟧ x → φ x)
  ax-Standard-3 : ∀ {ℓ} → {A : Set ℓ} → (φ : A → ESet) →
    (∀ x → st x → φ x → ⟦ φ ⟧ x)


-- AXIOM: Idealization --

postulate
  ax-Ideal-1 : {ℓ₁ ℓ₂ ℓ₃ : Level} {A : Set ℓ₁} {B : Set ℓ₂} (φ : A → B → Set ℓ₃) →
               (∀ (xs : List A) → st xs → ∃ λ b → ∀ (x : A) → x ∈ xs → φ x b) →
               ∃* λ b → ∀ (x : A) → st x → φ x b
  ax-Ideal-2 : {ℓ₁ ℓ₂ ℓ₃ : Level} {A : Set ℓ₁} {B : Set ℓ₂} → (φ : A → B → Set ℓ₃) →
               (∃* λ b → ∀ (x : A) → st x → φ x b) →
               ∀ (xs : List A) → st xs → ∃ λ b → ∀ (x : A) → x ∈ xs → φ x b

--------------------------------------------------------------------------------


{-# OPTIONS --omega-in-omega --no-pattern-matching #-}

module IST.Util where

-- We prove a bunch of useful lemmata.

open import Agda.Primitive
open import IST.Safe.Util public

open import IST.Base

-- If x and y are standard, then so is (x , y).
lemma-pairing : {ℓ₁ ℓ₂ : Level} {A : Set ℓ₁} {B : Set ℓ₂} → (x : A) → (y : B) →
                st {ℓ₁} {A} x → st {ℓ₂} {B} y → st {ℓ₁ ⊔ ℓ₂} {A ∧ B} (x , y)
lemma-pairing {ℓ₁} {ℓ₂} {A} {B} x y st-x st-y = st-pair-x-y where
  pair : A → B → A ∧ B
  pair = _,_
  st-pair : st pair
  st-pair = st-∃-_,_-full
  st-pair-x : st (pair x)
  st-pair-x = st-fun A (B → A ∧ B) pair x st-pair st-x
  st-pair-x-y : st (pair x y)
  st-pair-x-y = st-fun B (A ∧ B) (pair x) y st-pair-x st-y

lemma-proj₁ : {ℓ₁ ℓ₂ : Level} {A : Set ℓ₁} {B : Set ℓ₂} → (ab : A ∧ B) → st ab → st (proj₁ ab)
lemma-proj₁ {ℓ₁} {ℓ₂} {A} {B} ab st-ab = st-sproj-ab where
  sproj : A ∧ B → A
  sproj = proj₁
  st-sproj : st sproj
  st-sproj = st-∃-proj₁-full
  st-sproj-ab : st (sproj ab)
  st-sproj-ab = st-fun _ _ sproj ab st-sproj st-ab
```

```
lemma-proj₁-d : {ℓ₁ ℓ₂ : Level} {A : Set ℓ₁} {B : A → Set ℓ₂} → (ab : ∃ λ a → B a) → st ab → st
(proj₁ ab)
lemma-proj₁-d {ℓ₁} {ℓ₂} {A} {B} ab st-ab = st-sproj-ab where
  sproj : (∃ λ a → B a) → A
  sproj = proj₁
  st-sproj : st sproj
  st-sproj = st-∃-proj₁-full
  st-sproj-ab : st (sproj ab)
  st-sproj-ab = st-fun _ _ sproj ab st-sproj st-ab

lemma-proj₂ : {ℓ₁ ℓ₂ : Level} {A : Set ℓ₁} {B : Set ℓ₂} → (ab : A ∧ B) → st ab → st (proj₂ ab)
lemma-proj₂ {ℓ₁} {ℓ₂} {A} {B} ab st-ab = st-sproj-ab where
  sproj : A ∧ B → B
  sproj = proj₂
  st-sproj : st sproj
  st-sproj = st-∃-proj₂-full
  st-sproj-ab : st (sproj ab)
  st-sproj-ab = st-fun _ _ sproj ab st-sproj st-ab

-- If b is standard, then so is any constant function returning b.
lemma-constfun : {ℓ₁ ℓ₂ : Level} → {A : Set ℓ₁} → {B : Set ℓ₂} → (b : B) → st b → st (λ (a : A) →
b)
lemma-constfun {_} {_} {A} {B} b st-b = st-K-b where
  K : B → A → B
  K x y = x
  st-K : st K
  st-K = st-abs-K B A
  st-K-b : st (K b)
  st-K-b = st-fun _ _ K b st-K st-b

lemma-constfun-h : {ℓ₁ ℓ₂ : Level} → {A : Set ℓ₁} → {B : Set ℓ₂} → (b : B) → st b → st (λ {a : A}
→ b)
lemma-constfun-h {ℓ₁} {ℓ₂} {A} {B} b st-b = st-K-b where
  K : B → {a : A} → B
  K x {y} = x
  st-K : st K
  st-K = st-abs-K-h B A
  st-K-b : st (λ {a : A} → K b {a})
  st-K-b = st-fun _ _ K b st-K st-b

--------------------------------------------------------------------------------



{-# OPTIONS --omega-in-omega --no-pattern-matching #-}

module IST.Naturals where

open import Agda.Primitive
open import IST.Safe.Naturals public

open import IST.Base
open SafeImportTools

st-ℕ : st {lsuc lzero} ℕ
st-ℕ = declared-in-safe-module ℕ

st-zero : st zero
st-zero = declared-in-safe-module zero

st-suc : st suc
st-suc = declared-in-safe-module suc

st-ℕ-induction : {ℓ : Level} → st λ {φ} → ℕ-induction {ℓ} {φ}
st-ℕ-induction {ℓ} = declared-in-safe-module λ {φ} → ℕ-induction {ℓ} {φ}

st-ℕ-induction-full : {ℓ : Level} → (φ : ℕ → Set ℓ) → st (ℕ-induction {ℓ} {φ})
st-ℕ-induction-full _ = declared-in-safe-module ℕ-induction

st-≤ : st _≤_
st-≤ = declared-in-safe-module _≤_

external-induction : {φ : ℕ → ESet} → φ zero → (∀ k → st k → φ k → φ (suc k)) → ∀ n → st n → φ
n
external-induction {φ} base-case inductive-case n st-n =
```

```
  ax-Standard-2 φ n st-n (ψ-forall n) where
  ψ : ℕ → Set
  ψ = ⟦ φ ⟧
  st-ψ : st ψ
  st-ψ = ax-Standard-1 φ
  ψ-base : ψ zero
  ψ-base = ax-Standard-3 φ zero st-zero base-case
  ψ-inductive-st : ∀ k → st k → ψ k → ψ (suc k)
  ψ-inductive-st k st-k ψ-k =
    ax-Standard-3 φ (suc k) (st-fun _ _ suc k st-suc st-k) (inductive-case k st-k (ax-Standard-2
φ k st-k ψ-k))
  ψ-inductive : ∀ k → ψ k → ψ (suc k)
  ψ-inductive = ax-Transfer-EI (∀' ℕ (λ k → int' (ψ k → ψ (suc k))))
    (λ k st-k → fromInternal (ψ-inductive-st k st-k))
    (st-ℕ , λ a st-a → st-→ (⟦ φ ⟧ a) (st-fun _ _ ψ a st-ψ st-a)
    (⟦ φ ⟧ (suc a)) (st-fun _ _ ψ (suc a) st-ψ (st-fun _ _ suc a st-suc st-a)))
  ψ-forall : ∀ n → ψ n
  ψ-forall = ℕ-induction ψ-base ψ-inductive

bounded-st : ∀ (b : ℕ) → st b → ∀ (n : ℕ) → n ≤ b → st n
bounded-st = external-induction {λ b → ∀ m → m ≤ b → st m} base-case inductive-case where
  base-case : ∀ m → m ≤ zero → st m
  base-case m m≤0 = transport* (sym (≤-than-zero m m≤0)) {λ n → st {lzero} {ℕ} n} st-zero
  inductive-case : ∀ k → st k → (∀ m → m ≤ k → st m) → ∀ n → n ≤ suc k → st n
  inductive-case k st-k inductive-hypothesis n n≤k+1 =
    by-cases* {lzero} {lzero} {n ≤ k} (st n) case-A case-B (≤-match n k n≤k+1) where
      case-A : n ≤ k → st n
      case-A = inductive-hypothesis n
      st-k+1 : st (suc k)
      st-k+1 = st-fun _ _ suc k st-suc st-k
      case-B : n ≡ suc k → st n
      case-B n-equals-k+1 = transport* (sym n-equals-k+1) {λ n → st {lzero} {ℕ} n} st-k+1

--------------------------------------------------------------------------------


{-# OPTIONS --omega-in-omega --no-pattern-matching #-}

module IST.FiniteSets where

open import Agda.Primitive
open import IST.Safe.FiniteSets public

open import IST.Base
open SafeImportTools

st-FiniteSet : st FiniteSet
st-FiniteSet = declared-in-safe-module FiniteSet

st-FiniteSet-Carrier : st FiniteSet.Carrier
st-FiniteSet-Carrier = declared-in-safe-module FiniteSet.Carrier

--------------------------------------------------------------------------------


{-# OPTIONS --omega-in-omega --no-pattern-matching #-}

module IST.Reals where

open import IST.Safe.Reals public

open import IST.Base
open SafeImportTools

st-ℝ : st ℝ
st-ℝ = declared-in-safe-module ℝ

st-+ : st _+_
st-+ = declared-in-safe-module _+_

st-minus : st minus
st-minus = declared-in-safe-module minus

st-· : st _·_
st-· = declared-in-safe-module _·_
```

```
st-inv : st inv
st-inv = declared-in-safe-module inv

st-< : st _<_
st-< = declared-in-safe-module _<_

st-≤ᵣ : st _≤ᵣ_
st-≤ᵣ = declared-in-safe-module _≤ᵣ_

st-0r : st 0r
st-0r = declared-in-safe-module 0r

st-1r : st 1r
st-1r = declared-in-safe-module 1r

st-inv-v : ∀ x → (e : x ≠ 0r) → st x → st (inv x e)
st-inv-v x e _ = declared-in-safe-module (inv x e)

st-2r : st 2r
st-2r = st-fun _ _ (_+_ 1r) 1r (st-fun _ _ _+_ 1r st-+ st-1r) st-1r

st-1/2r : st 1/2r
st-1/2r = st-inv-v 2r (λ _ → pos-2r) st-2r

st-/2r-v : (x : ℝ) → st x → st (x /2r)
st-/2r-v x st-x = st-fun _ _ (_·_ 1/2r) x (st-fun _ _ _·_ 1/2r st-· st-1/2r) st-x

--------------------------------------------------------------------------------


{-# OPTIONS --omega-in-omega --no-pattern-matching #-}

module IST.Groups where

open import IST.Safe.Groups public

open import IST.Base
open SafeImportTools

st-Group : st Group
st-Group = declared-in-safe-module Group

st-Group-Carrier : st Group.Carrier
st-Group-Carrier = declared-in-safe-module Group.Carrier

st-Group-identity : st Group.identity
st-Group-identity = declared-in-safe-module Group.identity

st-Group-operation : st Group.operation
st-Group-operation = declared-in-safe-module Group.operation

st-Group-inverse : st Group.inverse
st-Group-inverse = declared-in-safe-module Group.inverse

st-Group-power : st Group.power
st-Group-power = declared-in-safe-module Group.power

st-FiniteGroup : st FiniteGroup
st-FiniteGroup = declared-in-safe-module FiniteGroup

st-FiniteGroup-Carrier : st FiniteGroup.Carrier
st-FiniteGroup-Carrier = declared-in-safe-module FiniteGroup.Carrier

st-FiniteGroup-identity : st FiniteGroup.identity
st-FiniteGroup-identity = declared-in-safe-module FiniteGroup.identity

st-FiniteGroup-operation : st FiniteGroup.operation
st-FiniteGroup-operation = declared-in-safe-module FiniteGroup.operation

st-FiniteGroup-inverse : st FiniteGroup.inverse
st-FiniteGroup-inverse = declared-in-safe-module FiniteGroup.inverse

st-FiniteGroup-order : st FiniteGroup.order
st-FiniteGroup-order = declared-in-safe-module FiniteGroup.order
```

```
st-FiniteGroup-power : st FiniteGroup.power
st-FiniteGroup-power = declared-in-safe-module FiniteGroup.power


st-PeriodicGroup : st PeriodicGroup
st-PeriodicGroup = declared-in-safe-module PeriodicGroup

st-PeriodicGroup-Carrier : st PeriodicGroup.Carrier
st-PeriodicGroup-Carrier = declared-in-safe-module PeriodicGroup.Carrier

st-PeriodicGroup-identity : st PeriodicGroup.identity
st-PeriodicGroup-identity = declared-in-safe-module PeriodicGroup.identity

st-PeriodicGroup-operation : st PeriodicGroup.operation
st-PeriodicGroup-operation = declared-in-safe-module PeriodicGroup.operation

st-PeriodicGroup-inverse : st PeriodicGroup.inverse
st-PeriodicGroup-inverse = declared-in-safe-module PeriodicGroup.inverse

st-PeriodicGroup-order : st PeriodicGroup.order
st-PeriodicGroup-order = declared-in-safe-module PeriodicGroup.order

st-PeriodicGroup-power : st PeriodicGroup.power
st-PeriodicGroup-power = declared-in-safe-module PeriodicGroup.power

--------------------------------------------------------------------------------


{-# OPTIONS --omega-in-omega --no-pattern-matching #-}

module IST.MetricSpaces where

open import Agda.Primitive
open import IST.Safe.MetricSpaces public

open import IST.Base
open SafeImportTools

st-MetricSpace : st MetricSpace
st-MetricSpace = declared-in-safe-module MetricSpace

st-MetricSpace-Carrier : st MetricSpace.Carrier
st-MetricSpace-Carrier = declared-in-safe-module MetricSpace.Carrier

st-MetricSpace-distance : st MetricSpace.distance
st-MetricSpace-distance = declared-in-safe-module MetricSpace.distance

st-MetricSpace-Carrier-full : (M : MetricSpace) → st M → st (MetricSpace.Carrier M)
st-MetricSpace-Carrier-full M st-M = st-fun _ _ MetricSpace.Carrier M st-MetricSpace-Carrier st-
M

st-MetricSpace-distance-full : (M : MetricSpace) → st M → st (MetricSpace.distance M)
st-MetricSpace-distance-full M st-M = declared-in-safe-module (MetricSpace.distance M)

--------------------------------------------------------------------------------


module IST.GroupActions where

open import IST.Safe.GroupActions public

--------------------------------------------------------------------------------


{-# OPTIONS --omega-in-omega --no-pattern-matching #-}

module IST.NewmansTheorem where

open import Agda.Primitive
open import IST.Safe.NewmansTheorem public

open import IST.Base
open SafeImportTools

st-NewmanSpace : st NewmanSpace
st-NewmanSpace = declared-in-safe-module NewmanSpace
```

```
st-NewmanSpace-asMetricSpace : st NewmanSpace.asMetricSpace
st-NewmanSpace-asMetricSpace = declared-in-safe-module NewmanSpace.asMetricSpace

st-NewmanSpace-inhabitant : st NewmanSpace.inhabitant
st-NewmanSpace-inhabitant = declared-in-safe-module NewmanSpace.inhabitant

st-NewmanSpace-newman-constant : st NewmanSpace.newman-constant
st-NewmanSpace-newman-constant = declared-in-safe-module NewmanSpace.newman-constant

--------------------------------------------------------------------------------



{-# OPTIONS --omega-in-omega #-}

-- TODO: Make sure that this confirms to the new coding standards.
-- This is taken from an older version of the proof code.
-- Note that our main proof does not rely on these arguments.

module IST.Ultrafilters where

open import Agda.Primitive
open import IST.Base
open import IST.Util

⋂ : {I : Set} → List (I → Set) → I → Set
⋂ [] i = ⊤
⋂ (φ ∷ []) i = φ i
⋂ (φ ∷ φs) i = φ i ∧ (⋂ φs i)

lemma-⋂ : {I : Set} → (φs : List (I → Set)) → ∀ (i : I) → ⋂ φs i → ∀ φ → φ ∈ φs → φ i
lemma-⋂ (.φ ∷ []) i has-i φ ∈-head = has-i
lemma-⋂ (.φ ∷ (ψ ∷ φs)) i has-i φ ∈-head = proj₁ has-i
lemma-⋂ (ψ ∷ []) i has-i φ (∈-tail ())
lemma-⋂ (ψ₁ ∷ (ψ₂ ∷ ψs)) i has-i φ (∈-tail φ∈φs) = lemma-⋂ (ψ₂ ∷ ψs) i (proj₂ has-i) φ φ∈φs

_⊆_ : {I : Set} → List (I → Set) → ((I → Set) → Set) → Set
[] ⊆ UF = ⊤
(φ ∷ []) ⊆ UF = UF φ
(φ ∷ φs) ⊆ UF = UF φ ∧ (φs ⊆ UF)

_⇒_ : {I : Set} → (I → Set) → (I → Set) → Set
φ ⇒ ψ = ∀ i → φ i → ψ i

~ : {I : Set} → (I → Set) → I → Set
~ φ i = φ i → ⊥

module Stage1
  (I : Set)
  (st-I : st I)
  (UF : (I → Set) → Set)
  (UF-upward : {φ ψ : I → Set} → φ ⇒ ψ → UF φ → UF ψ)
  (UF-inhabit : {φ : I → Set} → UF φ → ∃ λ i → φ i)
  (UF-fip : {φs : List (I → Set)} → φs ⊆ UF → UF (⋂ φs))
  (UF-alt : {φ : I → Set} → (UF φ → ⊥) → UF (~ φ))
  where
  ∅ : I → Set
  ∅ i = ⊥

  U : I → Set
  U i = ⊤

  step-1 : UF ∅ → ⊥
  step-1 has-∅ = proj₂ (UF-inhabit has-∅)

  step-2 : UF U
  step-2 = UF-upward {~ ∅} {U} (λ i _ → tt) (UF-alt step-1)

  arbitrary : I
  arbitrary = proj₁ (UF-inhabit step-2)

  Element : Set₁
  Element = ∃ λ (φ : I → Set) → UF φ
```

```
    reduce : List Element → List (I → Set)
    reduce [] = []
    reduce (φ ∷ φs) = (proj₁ φ) ∷ reduce φs

    lemma-reduce : (φs : List Element) → reduce φs ⊆ UF
    lemma-reduce [] = tt
    lemma-reduce (φ ∷ []) = proj₂ φ
    lemma-reduce (φ ∷ (ψ ∷ φs)) = proj₂ φ , lemma-reduce (ψ ∷ φs)

    step-3 : ∀ (φs : List Element) → st φs → ∃ λ (i : I) → ∀ (φ : Element) → φ ∈ φs → proj₁ φ i
    step-3 φs _ = proj₁ ⋂-inhabit , λ φ φ∈φs → jump (proj₁ φ) (lemma φ φs φ∈φs) where
      ⋂-inhabit : ∃ λ (i : I) → ⋂ (reduce φs) i
      ⋂-inhabit = UF-inhabit (UF-fip {reduce φs} (lemma-reduce φs))
      jump : ∀ (φ : I → Set) → φ ∈ reduce φs → φ (proj₁ ⋂-inhabit)
      jump = lemma-⋂ (reduce φs) (proj₁ ⋂-inhabit) (proj₂ ⋂-inhabit)
      lemma : (φ : Element) → (φs : List Element) → φ ∈ φs → proj₁ φ ∈ reduce φs
      lemma φ .(φ ∷ _) ∈-head = ∈-head
      lemma φ (ψ ∷ φs) (∈-tail p) = ∈-tail (lemma φ φs p)

    thm-1 : ∃* λ (i : I) → ∀ (φ : Element) → st φ → proj₁ φ i
    thm-1 = ax-Ideal-1 _ step-3

    ω : I
    ω = proj₁ thm-1

    module Stage2
      (st-UF : st UF)
      (UF-2val* : (φ : ESet) → (A : I → Set) → (UF A ≡ ⊤ → φ) → (UF A ≡ ⊥ → φ) → φ)
      where

      _~UF~_ : {A : I → Set} → (∀ (i : I) → A i) → (∀ (i : I) → A i) → Set
      f ~UF~ g = UF (λ i → f i ≡ g i)

      _~ω~_ : {A : I → Set} → (∀ (i : I) → A i) → (∀ (i : I) → A i) → Set
      f ~ω~ g = f ω ≡ g ω

      thm-2 : {A : I → Set} → st A → (f : ∀ (i : I) → A i) → st f → (g : ∀ (i : I) → A i) → st g →
              f ~UF~ g → f ~ω~ g
      thm-2 {A} st-A f st-f g st-g p = using-thm-1 where
        st-f=g : st (λ i → f i ≡ g i)
        st-f=g = st-req-f-g where
          recombinator : ({X : Set} → X → X → Set) → (b : I → Set) → (f : (i : I) → b i) → (g : (i
: I) → b i) → I → Set
          recombinator = λ (a : {X : Set} → X → X → Set) → λ (b : I → Set) → λ (f : (i : I) → b i)
→ λ (g : (i : I) → b i) →
                          λ (i : I) → a {b i} (f i) (g i)
          st-recombinator : st recombinator
          st-recombinator = st-abs-5 I
          recombinator-≡ : (b : I → Set) → (f : (i : I) → b i) → (g : (i : I) → b i) → I → Set
          recombinator-≡ = recombinator (_≡_ {lzero})
          st-recombinator-≡ : st recombinator-≡
          st-recombinator-≡ = st-fun _ _ recombinator (_≡_ {lzero}) st-recombinator st-≡
          req : (f : ∀ i → A i) → (g : ∀ i → A i) → I → Set
          req = recombinator-≡ A
          st-req : st req
          st-req = st-fun-d _ _ recombinator-≡ A st-recombinator-≡ st-A
          req-f : (g : ∀ i → A i) → I → Set
          req-f = req f
          st-req-f : st req-f
          st-req-f = st-fun _ _ req f st-req st-f
          req-f-g : I → Set
          req-f-g = req-f g
          st-req-f-g : st req-f-g
          st-req-f-g = st-fun _ _ req-f g st-req-f st-g
        eq : I → Set
        eq i = f i ≡ g i
        pair : (A : I → Set) → UF A → Element
        pair A a = A , a
        st-pair : st pair
        st-pair = st-∃-_,_-full
        st-pair-eq : st (pair eq)
        st-pair-eq = st-fun-d _ _ pair eq st-pair st-f=g
        st-pair-eq-p : st (pair eq p)
        st-pair-eq-p = st-fun-d _ _ (pair eq) p st-pair-eq (st-UF-p p) where
```

```
        if-⊥ : UF eq ≡ ⊥ → ∀ (p : UF eq) → st p
        if-⊥ x = transport* (sym x) {λ S → ∀ (p : S) → st p} λ ()
        if-T : UF eq ≡ T → ∀ (p : UF eq) → st p
        if-T x = transport* (sym x) {λ S → ∀ (p : S) → st p} helper where
          helper : (p : T) → st p
          helper tt = st-tt
        st-UF-p : ∀ (p : UF eq) → st p
        st-UF-p = UF-2val* (∀ (p : UF eq) → st p) eq if-T if-⊥
      using-thm-1 : f ω ≡ g ω
      using-thm-1 = proj₂ thm-1 (eq , p) st-pair-eq-p

-- we get the converse of thm-2 by the exact same argument, as ¬(f ~UF~ g) → UF (λ i → ¬ (f i ≡
g i))

--------------------------------------------------------------------------------


{-# OPTIONS --omega-in-omega --no-pattern-matching #-}

module IST.PredicatedTopologies where

open import Agda.Primitive
open import IST.Base
open import IST.Reals

----
-- Def. A relational space consists of a carrier set C and a reflexive
-- binary predicate (the 'nearness predicate') on C.

record IsPredicatedSpace
  (Carrier : Set)
  (nearby : Carrier → Carrier → ESet)
  : ESet where
  field
    reflexive : ∀ x → nearby x x

record PredicatedSpace : ESet₁ where
  field
    Carrier : Set
    nearby : Carrier → Carrier → ESet
    isPredicatedSpace : IsPredicatedSpace Carrier nearby
  open IsPredicatedSpace isPredicatedSpace public


-- Def. A separable space is a relational space where no two standard points
-- are neighbors. (normally known as T1 space, we refer to those as Kolmogorov)

record IsSeparableSpace
  (Carrier : Set)
  (nearby : Carrier → Carrier → ESet)
  : ESet where
  field
    isPredicatedSpace : IsPredicatedSpace Carrier nearby
    separable : ∀ x → st x → ∀ y → st y → nearby x y → nearby y x → x ≡ y

record SeparableSpace : ESet₁ where
  field
    Carrier : Set
    nearby : Carrier → Carrier → ESet
    isSeparableSpace : IsSeparableSpace Carrier nearby
  open IsSeparableSpace isSeparableSpace public
  open IsPredicatedSpace isPredicatedSpace public


-- Def. A compact space is a relation space where every every element is near
-- a standard element.

record IsCompactSpace
  (Carrier : Set)
  (nearby : Carrier → Carrier → ESet)
  : ESet where
  field
    isPredicatedSpace : IsPredicatedSpace Carrier nearby
    compact : ∀ x → ∃* λ y → st y *∧* nearby y x
```

```
record CompactSpace : ESet₁ where
  field
    Carrier : Set
    nearby : Carrier → Carrier → ESet
    isCompactSpace : IsCompactSpace Carrier nearby
  open IsCompactSpace isCompactSpace public
  open IsPredicatedSpace isPredicatedSpace public


-- Def. A Hausdorff space is a relational space where two different standard
-- points do not share a neighbor.

record IsHausdorffSpace
  (Carrier : Set)
  (nearby : Carrier → Carrier → ESet)
  : ESet where
  field
    isPredicatedSpace : IsPredicatedSpace Carrier nearby
    hausdorff : ∀ x → st x → ∀ y → st y → ∀ z → nearby x z → nearby y z → x ≡ y

record HausdorffSpace : ESet₁ where
  field
    Carrier : Set
    nearby : Carrier → Carrier → ESet
    isHausdorffSpace : IsHausdorffSpace Carrier nearby
  open IsHausdorffSpace isHausdorffSpace public
  open IsPredicatedSpace isPredicatedSpace public
  -- Thm. Every Hausdorff space is separable.
  private
    separable : ∀ x → st x → ∀ y → st y → nearby x y → nearby y x → x ≡ y
    separable x st-x y st-y x-near-y y-near-x =
      hausdorff x st-x y st-y x (reflexive x) y-near-x
    isSeparableSpace : IsSeparableSpace Carrier nearby
    isSeparableSpace = record { isPredicatedSpace = isPredicatedSpace; separable = separable }
  open IsSeparableSpace isSeparableSpace public


-- Def. A compact Hausdorff space is a relational space that is also a compact
-- space. Duh.

record IsCompactHausdorffSpace
  (Carrier : Set)
  (nearby : Carrier → Carrier → ESet)
  : ESet where
  field
    isHausdorffSpace : IsHausdorffSpace Carrier nearby
    isCompactSpace : IsCompactSpace Carrier nearby

record CompactHausdorffSpace : ESet₁ where
  field
    Carrier : Set
    nearby : Carrier → Carrier → ESet
    isHausdorffSpace : IsHausdorffSpace Carrier nearby
    isCompactSpace : IsCompactSpace Carrier nearby
  open IsHausdorffSpace isHausdorffSpace public
  open IsPredicatedSpace isPredicatedSpace public
  open IsCompactSpace isCompactSpace public


-- Def. An equivalence space is a relational space whose
-- nearness predicate is transitive and symmetric.

record IsEquivalenceSpace
  (Carrier : Set)
  (nearby : Carrier → Carrier → ESet)
  : ESet where
  field
    isPredicatedSpace : IsPredicatedSpace Carrier nearby
    transitive : ∀ x y z → nearby x y → nearby y z → nearby x z
    symmetric : ∀ x y → nearby x y → nearby y x

record EquivalenceSpace : ESet₁ where
  field
    Carrier : Set
    nearby : Carrier → Carrier → ESet
    isEquivalenceSpace : IsEquivalenceSpace Carrier nearby
```

```
  open IsEquivalenceSpace isEquivalenceSpace public
  open IsPredicatedSpace isPredicatedSpace public


-- Def. A Hausdorff equivalence space is an equivalence space that is
-- also a Hausdorff space. Duh.

record IsHausdorffEquivalenceSpace
  (Carrier : Set)
  (nearby : Carrier → Carrier → ESet)
  : ESet where
  field
    isHausdorffSpace : IsHausdorffSpace Carrier nearby
    isEquivalenceSpace : IsEquivalenceSpace Carrier nearby

record HausdorffEquivalenceSpace : ESet₁ where
  field
    Carrier : Set
    nearby : Carrier → Carrier → ESet
    isHausdorffSpace : IsHausdorffSpace Carrier nearby
    isEquivalenceSpace : IsEquivalenceSpace Carrier nearby
  open IsHausdorffSpace isHausdorffSpace public
  open IsPredicatedSpace isPredicatedSpace public
  open IsEquivalenceSpace isEquivalenceSpace public


-- Def. A compact Hausdorff equivalence space is an equivalence space that is also a compact
-- space. Duh.

record IsCompactHausdorffEquivalenceSpace
  (Carrier : Set)
  (nearby : Carrier → Carrier → ESet)
  : ESet where
  field
    isHausdorffSpace : IsHausdorffSpace Carrier nearby
    isCompactSpace : IsCompactSpace Carrier nearby
    isEquivalenceSpace : IsEquivalenceSpace Carrier nearby

record CompactHausdorffEquivalenceSpace : ESet₁ where
  field
    Carrier : Set
    nearby : Carrier → Carrier → ESet
    isHausdorffSpace : IsHausdorffSpace Carrier nearby
    isCompactSpace : IsCompactSpace Carrier nearby
    isEquivalenceSpace : IsEquivalenceSpace Carrier nearby
  open IsHausdorffSpace isHausdorffSpace public
  open IsPredicatedSpace isPredicatedSpace public
  open IsCompactSpace isCompactSpace public
  open IsEquivalenceSpace isEquivalenceSpace public


open import IST.MetricSpaces

-- Thm. Every standard metric space induces a Hausdorff equivalence space by setting
-- x o- y ↔ ∀ˢ ε > 0. d(x,y) < ε
metric-to-hausdorff-equivalence : (MS : MetricSpace) → st MS → HausdorffEquivalenceSpace
metric-to-hausdorff-equivalence MS st-MS =
  record { Carrier = M
         ; nearby = nearby
         ; isHausdorffSpace = isHausdorffSpace
         ; isEquivalenceSpace = isEquivalenceSpace
         } where
  M : Set
  M = MetricSpace.Carrier MS

  st-M : st M
  st-M = st-MetricSpace-Carrier-full MS st-MS

  d : M → M → ℝ
  d = MetricSpace.distance MS

  st-d : st d
  st-d = st-MetricSpace-distance-full MS st-MS

  nearby : M → M → ESet
  nearby x y = ∀ (ε : ℝ) → st ε → 0r < ε → d x y < ε
```

```
  reflexive : ∀ x → nearby x x
  reflexive x ε st-ε 0r<ε = dxx<ε where
    open MetricSpace MS
    dxx<ε : d x x < ε
    dxx<ε = transport (sym (reflexive-2 x)) {λ z → z < ε} 0r<ε

  symmetric : ∀ x y → nearby x y → nearby y x
  symmetric x y x-near-y ε st-ε pos-ε = dyx<ε where
    open MetricSpace MS
    dxy<ε : d x y < ε
    dxy<ε = x-near-y ε st-ε pos-ε
    dyx<ε : d y x < ε
    dyx<ε = transport (symmetry x y) {λ p → p < ε} dxy<ε

  transitive : ∀ x y z → nearby x y → nearby y z → nearby x z
  transitive x y z x-near-y y-near-z ε st-ε pos-ε = dxz' where
    open MetricSpace MS
    ε/2 : ℝ
    ε/2 = ε /2r
    st-ε/2 : st ε/2
    st-ε/2 = st-/2r-v ε st-ε
    pos-ε/2 : 0r < ε/2
    pos-ε/2 = pos-/2r-v ε pos-ε
    dxy : d x y < ε/2
    dxy = x-near-y ε/2 st-ε/2 pos-ε/2
    dyz : d y z < ε/2
    dyz = y-near-z ε/2 st-ε/2 pos-ε/2
    dxy-dyx : d x y + d y z < ε/2 + ε/2
    dxy-dyx = <-tran _ _ _ step-1 step-2 where
      step-1 : d x y + d y z < ε/2 + d y z
      step-1 = <-plus (d x y) ε/2 (d y z) dxy
      step-2 : ε/2 + d y z < ε/2 + ε/2
      step-2 = transport +-comm {λ p → p < ε/2 + ε/2} (<-plus (d y z) ε/2 ε/2 dyz)
    dxz : d x z < ε/2 + ε/2
    dxz = triangle x y z (ε/2 + ε/2) dxy-dyx
    dxz' : d x z < ε
    dxz' = transport /2r-half {λ p → d x z < p} dxz

  isPredicatedSpace : IsPredicatedSpace M nearby
  isPredicatedSpace = record { reflexive = reflexive }

  isEquivalenceSpace : IsEquivalenceSpace M nearby
  isEquivalenceSpace =
    record { isPredicatedSpace = isPredicatedSpace
           ; transitive = transitive
           ; symmetric = symmetric
           }
  hausdorff : ∀ x → st x → ∀ y → st y → ∀ z → nearby x z → nearby y z → x ≡ y
  hausdorff x st-x y st-y z x-near-z y-near-z = reflexive-1 x y zero-dxy where
    open MetricSpace MS
    x-near-y : nearby x y
    x-near-y = transitive x z y x-near-z (symmetric y z y-near-z)
    x-near-y-int : (ε : ℝ) → st ε → internal (0r < ε → d x y < ε)
    x-near-y-int ε st-ε = fromInternal (x-near-y ε st-ε)
    st-dxy : st (d x y)
    st-dxy = st-fun _ _ (d x) y (st-fun _ _ d x st-d st-x) st-y
    Φ : TransferPred
    Φ = ∀' ℝ λ ε → int' (0r < ε → d x y < ε)
    std-Φ : st ℝ *∧* ((ε : ℝ) → st ε → st (0r < ε → d x y < ε))
    std-Φ = st-ℝ , (λ ε st-ε → st-→ (0r < ε)
                                    (st-fun _ _ (_<_ 0r) ε (st-fun _ _ _<_ 0r st-< st-0r) st-ε)
                                    (d x y < ε)
                                    (st-fun _ _ (_<_ (d x y)) ε (st-fun _ _ _<_ (d x y) st-< st-
dxy) st-ε))
    dxy<ε : ∀ (ε : ℝ) → 0r < ε → d x y < ε
    dxy<ε = ax-Transfer-EI Φ x-near-y-int std-Φ
    zero-dxy : d x y ≡ 0r
    zero-dxy = lemma-ε-of-room (d x y) dxy<ε (nonnegative x y)

  isHausdorffSpace : IsHausdorffSpace M nearby
  isHausdorffSpace =
    record { isPredicatedSpace = isPredicatedSpace
           ; hausdorff = hausdorff
           }
```

```
--------------------------------------------------------------------------------


{-# OPTIONS --omega-in-omega --no-pattern-matching #-}

module IST.Approximation where

open import Agda.Primitive
open import IST.Base
open import IST.PredicatedTopologies


record IsApproximation
  (Source : Set)
  (Target : Set)
  (Map : Source → Target → ESet)
  : ESet where
  field
    Target-st : st Target
    Map-exists : ∀ (g : Target) → st g → ∃* λ (h : Source) → Map h g
    Map-unique-Source :
      ∀ (g : Target) → st g →
      ∀ (h₁ : Source) → Map h₁ g → ∀ (h₂ : Source) → Map h₂ g → h₁ ≡ h₂
    Map-unique-Target :
      ∀ (g₁ : Target) → st g₁ → ∀ (g₂ : Target) → st g₂ →
      ∀ (h : Source) → Map h g₁ → Map h g₂ → g₁ ≡ g₂

-- Map-cont :
--   ∀ (h₁ h₂ : Source) → ∀ (g₁ g₂ : Target) → Map h₁ g₁ → Map h₂ g₂ →
--   nearby h₁ h₂ → nearby g₁ g₂
-- -- makes no sense since S-continuity relies on the standardness
-- -- of the first element of the nearness relation.

record Approximation (Source : Set) (Target : Set) : ESet₁ where
  field
    Map : Source → Target → ESet
    isApproximation : IsApproximation Source Target Map
  open IsApproximation isApproximation public


record IsTopApproximation
  (Source : PredicatedSpace)
  (Target : SeparableSpace)
  (Map : PredicatedSpace.Carrier Source → SeparableSpace.Carrier Target → ESet)
  : ESet where
  open SeparableSpace Target renaming
    ( Carrier to G
    ; nearby to G-near
    )
  open PredicatedSpace Source renaming
    ( Carrier to H
    ; nearby to H-near
    )
  field
    Target-st : st G
    Map-exists : ∀ (g : G) → st g → ∃* λ (h : H) → Map h g
    Map-Source :
      ∀ (g : G) → st g →
      ∀ (h₁ : H) → Map h₁ g → ∀ (h₂ : H) → Map h₂ g → H-near h₁ h₂
    Map-Target :
      ∀ (g₁ : G) → st g₁ → ∀ (g₂ : G) → st g₂ →
      ∀ (h : H) → Map h g₁ → Map h g₂ → G-near g₁ g₂


record TopApproximation (Source : PredicatedSpace) (Target : SeparableSpace) : ESet₁ where
  field
    Map : PredicatedSpace.Carrier Source → SeparableSpace.Carrier Target → ESet
    isTopApproximation : IsTopApproximation Source Target Map
  open IsTopApproximation isTopApproximation public


open import IST.Groups

record IsFiniteGroupApproximation
```

```
    (Source : FiniteGroup)
    (Target : Group)
    (Map : FiniteGroup.Carrier Source → Group.Carrier Target → ESet)
    : ESet where
    field
      isApproximation : IsApproximation (FiniteGroup.Carrier Source) (Group.Carrier Target) Map
      Map-homomorphism :
        ∀ (h₁ h₂ : FiniteGroup.Carrier Source) →
        ∀ (g₁ : Group.Carrier Target) → st g₁ → ∀ (g₂ : Group.Carrier Target) → st g₂ →
        Map h₁ g₁ → Map h₂ g₂ → Map (FiniteGroup.operation Source h₁ h₂) (Group.operation Target g₁
g₂)
    open IsApproximation isApproximation
    open Group Target renaming
      ( Carrier to G
      ; identity to 1G
      ; operation to xG
      ; inverse to iG
      ; assoc to G-associative
      ; unit-left to G-unit-left
      ; unit-right to G-unit-right
      ; inverse-left to G-inverse-left
      ; inverse-right to G-inverse-right
      )
    open FiniteGroup Source renaming
      ( Carrier to H
      ; identity to 1H
      ; operation to xH
      ; inverse to iH
      ; assoc to H-associative
      ; unit-left to H-unit-left
      ; unit-right to H-unit-right
      ; inverse-left to H-inverse-left
      ; inverse-right to H-inverse-right
      )
    Map-preserves-unit : st Target → Map 1H 1G
    Map-preserves-unit st-Target = Map-1H-1G where
      st-1G : st 1G
      st-1G = st-fun-d _ _ Group.identity Target st-Group-identity st-Target
      1H-unique : ∀ (h : H) → Map h 1G → h ≡ 1H
      1H-unique h Map-h-1G = step-8 where
        step-1 : Map (xH h h) (xG 1G 1G)
        step-1 = Map-homomorphism h h 1G st-1G 1G st-1G Map-h-1G Map-h-1G
        step-2 : Map (xH h h) 1G
        step-2 = transport* (G-unit-left 1G) {λ z → Map (xH h h) z} step-1
        step-3 : xH h h ≡ h
        step-3 = Map-unique-Source 1G st-1G (xH h h) step-2 h Map-h-1G
        step-4 : xH (iH h) (xH h h) ≡ xH (iH h) h
        step-4 = cong (λ z → xH (iH h) z) step-3
        step-5 : xH (xH (iH h) h) h ≡ xH (iH h) (xH h h)
        step-5 = H-associative (iH h) h h
        step-6 : xH 1H h ≡ xH (xH (iH h) h) h
        step-6 = sym (cong (λ z → xH z h) (H-inverse-left h))
        step-7 : h ≡ xH (xH (iH h) h) h
        step-7 = tran (sym (H-unit-left h)) step-6
        step-8 : h ≡ 1H
        step-8 = tran (tran (tran step-7 step-5) step-4) (H-inverse-left h)
      1H'-exists : ∃* λ (h : H) → Map h 1G
      1H'-exists = Map-exists 1G st-1G
      1H' : H
      1H' = proj₁ (Map-exists 1G st-1G)
      Map-1H'-1G : Map 1H' 1G
      Map-1H'-1G = proj₂ 1H'-exists
      1H'-equals-1H : 1H' ≡ 1H
      1H'-equals-1H = 1H-unique 1H' Map-1H'-1G
      Map-1H-1G : Map 1H 1G
      Map-1H-1G = transport* 1H'-equals-1H {λ z → Map z 1G} Map-1H'-1G
    Map-preserves-unit-Target : st Target → ∀ (g : G) → st g → Map 1H g → g ≡ 1G
    Map-preserves-unit-Target st-Target g st-g Map-1H-g =
      Map-unique-Target g st-g 1G st-1G 1H Map-1H-g (Map-preserves-unit st-Target) where
      st-1G : st 1G
      st-1G = st-fun-d _ _ Group.identity Target st-Group-identity st-Target

  record FiniteGroupApproximation (Source : FiniteGroup) (Target : Group) : ESet₁ where
    field
      Map : FiniteGroup.Carrier Source → Group.Carrier Target → ESet
```

```
    isFiniteGroupApproximation : IsFiniteGroupApproximation Source Target Map
  open IsFiniteGroupApproximation isFiniteGroupApproximation public
  open IsApproximation isApproximation public

--------------------------------------------------------------------------------


{-# OPTIONS --omega-in-omega --no-pattern-matching #-}

module IST.Results.ExtensionTheorem where

open import IST.Base
open import IST.Util
open import IST.Approximation
open import IST.PredicatedTopologies


-- Theorem: If H approximates G via ι, then we can extend every
-- function f : H → M (where M is a standard compact Hausdorff space) to a
-- function f' : G → M using standardization, setting
-- f' = ⟦ (g,m) ∈ G × M | ∃ h ∈ H. ι(h)=g ∧ f(h)=m ⟧.
record ExtensionTheorem : ESet where
  field
    G : Set
    H : Set
    A : Approximation H G
    M# : CompactHausdorffSpace
    st-M : st (CompactHausdorffSpace.Carrier M#)
    f : H → CompactHausdorffSpace.Carrier M#
  open CompactHausdorffSpace M# hiding (Carrier)
  open Approximation A
  private
    -- We refer to the underlying set of the space M# as M, and the
    -- approximation proper as ι.

    M : Set
    M = CompactHausdorffSpace.Carrier M#

    ι : H → G → ESet
    ι = Approximation.Map A

    -- Recall that by definition of approximation, G is standard.
    st-G : st G
    st-G = Approximation.Target-st A

    -- We construct the set f' = ⟦ ∃ˢ h. ι(h) = g ∧ m o- f(h) ⟧ by Standardization.
    pre-ext : G ∧ M → ESet
    pre-ext gm = ∃* λ (h : H) → ι h (proj₁ gm) *∧* nearby (proj₂ gm) (f h)

  -- Construction:
  -- The set f' forms the graph of the function we seek.
  f' : G ∧ M → Set
  f' = ⟦ pre-ext ⟧

  st-f' : st f'
  st-f' = ax-Standard-1 pre-ext

  private
    st-f'gm : (g : G) → (m : M) → st g → st m → st (f' (g , m))
    st-f'gm g m st-g st-m = st-fun (G ∧ M) Set f' (g , m) st-f' (lemma-pairing g m st-g st-m)

  -- We prove that for standard g, there is always some standard m such that (g,m) ∈ f'.
  f'-exists-st : ∀ (g : G) → st g → ∃* λ (m : M) → st m *∧* internal (f' (g , m))
  f'-exists-st g st-g = m , st-m , fromInternal f'-gm where
    -- Take a standard g, and pick an approximation h with ι(h)=g.
    h : H
    h = proj₁ (Map-exists g st-g)
    ι-h-g : ι h g
    ι-h-g = proj₂ (Map-exists g st-g)
    -- Compute f(h). Use the compactness of M to find a standard point
    -- near f(h).
    m : M
    m = proj₁ (compact (f h))
    st-m : st m
    st-m = proj₁ (proj₂ (compact (f h)))
```

```
  m-near-fh : nearby m (f h)
  m-near-fh = proj₂ (proj₂ (compact (f h)))
  -- Since ι(h)=g and m lies near f(h), by definition (g,m) belongs to f'.
  pre-ext-gm : pre-ext (g , m)
  pre-ext-gm = h , (ι-h-g , m-near-fh)
  st-gm : st (g , m)
  st-gm = lemma-pairing g m st-g st-m
  f'-gm : f' (g , m)
  f'-gm = ax-Standard-3 pre-ext _ st-gm pre-ext-gm

-- Existence conclusion:
-- By transfer, for any g, there is some m such that (g,m) ∈ f'.
f'-exists : ∀ (g : G) → ∃ λ (m : M) → f' (g , m)
f'-exists = ax-Transfer-EI Φ f'-exists-st st-params-Φ where
  Φ : TransferPred
  Φ = ∀' G λ g → ∃' M λ m → int' (f' (g , m))
  st-params-Φ : std-params Φ
  st-params-Φ = st-G , λ a st-a →
                st-M , λ e st-e → st-fun _ _ f' (a , e) st-f' (lemma-pairing a e st-a st-e)

private
  -- Now we prove for standard g the uniqueness of the m such that (g,m) ∈ f'.
  -- This proves that f' forms (the graph of) a function.
  f'-unique-st :
    ∀ (g : G)  → st g → ∀ (m₁ : M) → st m₁ → ∀ (m₂ : M) → st m₂ →
    f' (g , m₁) → f' (g , m₂) →
    m₁ ≡ m₂
  f'-unique-st g st-g m₁ st-m₁ m₂ st-m₂ f'-gm₁ f'-gm₂ = m₁-equals-m₂ where
    -- Since (g,mᵢ) are standard, they satisfy the defining formula of f',
    -- so we can find hᵢ near gᵢ such that mᵢ lies near f(hᵢ).
    -- First, we pick h₁.
    st-gm₁ : st (g , m₁)
    st-gm₁ = lemma-pairing g m₁ st-g st-m₁
    pre-ext-gm₁ : pre-ext (g , m₁)
    pre-ext-gm₁ = ax-Standard-2 pre-ext (g , m₁) st-gm₁ f'-gm₁
    h₁ : H
    h₁ = proj₁ pre-ext-gm₁
    ι-h₁-g : ι h₁ g
    ι-h₁-g = proj₁ (proj₂ pre-ext-gm₁)
    m₁-near-fh₁ : nearby m₁ (f h₁)
    m₁-near-fh₁ = proj₂ (proj₂ pre-ext-gm₁)
    -- Now, we pick h₂.
    st-gm₂ : st (g , m₂)
    st-gm₂ = lemma-pairing g m₂ st-g st-m₂
    pre-ext-gm₂ : pre-ext (g , m₂)
    pre-ext-gm₂ = ax-Standard-2 pre-ext (g , m₂) st-gm₂ f'-gm₂
    h₂ : H
    h₂ = proj₁ pre-ext-gm₂
    ι-h₂-g : ι h₂ g
    ι-h₂-g = proj₁ (proj₂ pre-ext-gm₂)
    m₂-near-fh₂ : nearby m₂ (f h₂)
    m₂-near-fh₂ = proj₂ (proj₂ pre-ext-gm₂)
    -- Now, h₁ and h₂ both approximate g, so by the approximation
    -- uniqueness clause, h₁ = h₂.
    h₁-equals-h₂ : h₁ ≡ h₂
    h₁-equals-h₂ = Map-unique-Source g st-g h₁ ι-h₁-g h₂ ι-h₂-g
    fh₁-equals-fh₂ : f h₁ ≡ f h₂
    fh₁-equals-fh₂ = cong f h₁-equals-h₂
    -- Since m₂ lies near f(h₂), and h₁ = h₂, we have that
    -- m₂ lies near f(h₁) as well.
    m₂-near-fh₁ : nearby m₂ (f h₁)
    m₂-near-fh₁ =
      transport* (sym fh₁-equals-fh₂) {λ z -> nearby m₂ z} m₂-near-fh₂
    -- But then m₁ and m₂ share a common neighbor, f(h₁). By the Hausdorff
    -- property, this implies m₁ = m₂.
    m₁-equals-m₂ : m₁ ≡ m₂
    m₁-equals-m₂ = hausdorff m₁ st-m₁ m₂ st-m₂ (f h₁) m₁-near-fh₁ m₂-near-fh₁

-- Uniqueness conclusion:
-- Since uniqueness holds for standard g, Transfer gives that it holds for arbitrary g.
-- Hence, the set f' forms the graph of a function.
f'-unique : ∀ (g : G) → ∀ (m₁ : M) → ∀ (m₂ : M) → f' (g , m₁) → f' (g , m₂) → m₁ ≡ m₂
```

```
  f'-unique = ax-Transfer-EI Φ
    (λ g st-g m₁ st-m₁ m₂ st-m₂ → fromInternal (f'-unique-st g st-g m₁ st-m₁ m₂ st-m₂)) st-params-
Φ where
    Φ : TransferPred
    Φ = ∀' G λ g → ∀' M λ m₁ → ∀' M λ m₂ → int' (f' (g , m₁) → f' (g , m₂) → m₁ ≡ m₂)
    st-params-Φ : std-params Φ
    st-params-Φ =
      st-G , λ g st-g →
      st-M , λ m₁ st-m₁ →
      st-M , λ m₂ st-m₂ → st-Φ g st-g m₁ st-m₁ m₂ st-m₂ where
      st-f'-gm₁ : (g : G) → st g → (m₁ : M) → st m₁ → st (f' (g , m₁))
      st-f'-gm₁ g st-g m₁ st-m₁ = st-fun _ _ f' (g , m₁) st-f' (lemma-pairing g m₁ st-g st-m₁)
      st-f'-gm₂ : (g : G) → st g → (m₂ : M) → st m₂ → st (f' (g , m₂))
      st-f'-gm₂ g st-g m₂ st-m₂ = st-fun _ _ f' (g , m₂) st-f' (lemma-pairing g m₂ st-g st-m₂)
      st-m₁≡m₂ : (m₁ : M) → st m₁ → (m₂ : M) → st m₂ → st (m₁ ≡ m₂)
      st-m₁≡m₂ m₁ st-m₁ m₂ st-m₂ = st-fun M Set (_≡_ m₁) m₂ (st-fun M (M → Set) _≡_ m₁ st-≡-full
st-m₁) st-m₂
      st-f'-gm₂-st-m₁≡m₂ : (g : G) → st g → (m₁ : M) → st m₁ → (m₂ : M) → st m₂ → st (f' (g , m₂)
→ m₁ ≡ m₂)
      st-f'-gm₂-st-m₁≡m₂ g st-g m₁ st-m₁ m₂ st-m₂ =
        st-→ _ (st-f'-gm₂ g st-g m₂ st-m₂) _ (st-m₁≡m₂ m₁ st-m₁ m₂ st-m₂)
      st-Φ : (g : G) → st g → (m₁ : M) → st m₁ → (m₂ : M) → st m₂ → st (f' (g , m₁) → f' (g , m₂)
→ m₁ ≡ m₂)
      st-Φ g st-g m₁ st-m₁ m₂ st-m₂ =
        st-→ (f' (g , m₁)) (st-f'-gm₁ g st-g m₁ st-m₁) (f' (g , m₂) → m₁ ≡ m₂)
             (st-f'-gm₂-st-m₁≡m₂ g st-g m₁ st-m₁ m₂ st-m₂)

{-
-- Theorem 2: If the sequence H approximates the structure G in the sense of
-- Zilber, then there is some H(ω) that approximates G in the sense above.
module Thm2
  (I : Set)
  (H : I → Set)
  (_~D~_ : (∀ i → H i) → (∀ i → H i) → Set)
  (st-D : st _~D~_)
  (ω : I)
  (ax-ω-1 : ∀ (f g : ∀ i → H i) → st f → st g → f ~D~ g → f ω ≡ g ω)
  (ax-ω-2 : ∀ (f g : ∀ i → H i) → st f → st g → f ω ≡ g ω → f ~D~ g)
  (G : Set)
  (st-G : st G)
  (φ : G → (∀ i → H i) → Set)
  (φ-exists : ∀ (g : G) → ∃ λ (h : ∀ i → H i) → φ g h)
  (lim : (∀ i → H i) → G)
  (lim-surjective : ∀ (g : G) → ∃ λ (h : ∀ i → H i) → lim h ≡ g)
  (lim-respects-D : ∀ (h₁ h₂ : ∀ i → H i) → h₁ ~D~ h₂ → lim h₁ ≡ lim h₂)
  (lim-preserves-φ : ∀ (g : G) → ∀ (h : ∀ i → H i) → φ g h → _≡_ g (lim h))
  where
  colim : G → (∀ i → H i)
  colim g = ∃.proj₁ (φ-exists g)

  colim-splits-lim : ∀ (g : G) → lim (colim g) ≡ g
  colim-splits-lim g = sym step-2 where
    step-1 : φ g (colim g)
    step-1 = ∃.proj₂ (φ-exists g)
    step-2 : g ≡ lim (colim g)
    step-2 = lim-preserves-φ g (colim g) step-1

  ι : H ω → G → Setω
  ι h g = internal (colim g ω ≡ h)

  ι-exists : ∀ (g : G) → st g → ∃* λ (h : H ω) → ι h g
  ι-exists g st-g = colim g ω , fromInternal refl

  ι-unique : ∀ (g : G) → st g → ∀ (h₁ : H ω) → ι h₁ g → ∀ (h₂ : H ω) → ι h₂ g → h₁ ≡ h₂
  ι-unique g st-g h₁ (fromInternal ι-h₁-g) h₂ (fromInternal ι-h₂-g) = tran (sym ι-h₁-g) ι-h₂-g

  open Thm1 G (H ω) st-G ι ι-exists ι-unique

-- If furthermore everything in Thm2 is standard, then we have co-uniquness as well.
module Thm2-X
  (I : Set)
  (H : I → Set)
  (_~D~_ : (∀ i → H i) → (∀ i → H i) → Set)
```

```
    (st-D : st _~D~_)
    (ω : I)
    (ax-ω-1 : ∀ (f g : ∀ i → H i) → st f → st g → f ~D~ g → f ω ≡ g ω)
    (ax-ω-2 : ∀ (f g : ∀ i → H i) → st f → st g → f ω ≡ g ω → f ~D~ g)
    (G : Set)
    (st-G : st G)
    (φ : G → (∀ i → H i) → Set)
    (φ-exists : ∀ (g : G) → ∃ λ (h : ∀ i → H i) → φ g h)
    (lim : (∀ i → H i) → G)
    (lim-surjective : ∀ (g : G) → ∃ λ (h : ∀ i → H i) → lim h ≡ g)
    (lim-respects-D : ∀ (h₁ h₂ : ∀ i → H i) → h₁ ~D~ h₂ → lim h₁ ≡ lim h₂)
    (lim-preserves-φ : ∀ (g : G) → ∀ (h : ∀ i → H i) → φ g h → _≡_ g (lim h))
    (φ-exists-st : ∀ (g : G) → st g → st (proj₁ (φ-exists g)) )
   where

     open Thm2 I H _~D~_ st-D ω ax-ω-1 ax-ω-2 G st-G φ φ-exists lim lim-surjective lim-respects-D
lim-preserves-φ

     st-colim-v : ∀ (g : G) → st g → st (colim g)
     st-colim-v g st-g = φ-exists-st g st-g

     ι-counique : ∀ (h : H ω) → ∀ (g₁ g₂ : G) → st g₁ → st g₂ → ι h g₁ → ι h g₂ → g₁ ≡ g₂
     ι-counique h g₁ g₂ st-g₁ st-g₂ ι-h-g₁ ι-h-g₂ = equality where
       step-1 : ι (colim g₁ ω) g₁
       step-1 with proj₂ (ι-exists g₁ st-g₁)
       step-1 | fromInternal x = fromInternal x
       step-2 : colim g₁ ω ≡ h
       step-2 = sym (ι-unique g₁ st-g₁ h ι-h-g₁ (colim g₁ ω) step-1)
       step-3 : ι (colim g₂ ω) g₂
       step-3 with proj₂ (ι-exists g₂ st-g₂)
       step-3 | fromInternal x = fromInternal x
       step-4 : colim g₂ ω ≡ h
       step-4 = sym (ι-unique g₂ st-g₂ h ι-h-g₂ (colim g₂ ω) step-3)
       step-5 : colim g₁ ω ≡ colim g₂ ω
       step-5 = tran step-2 (sym step-4)
       step-6 : colim g₁ ~D~ colim g₂
       step-6 = ax-ω-2 (colim g₁) (colim g₂) (st-colim-v g₁ st-g₁) (st-colim-v g₂ st-g₂) step-5
       step-7 : lim (colim g₁) ≡ lim (colim g₂)
       step-7 = lim-respects-D (colim g₁) (colim g₂) step-6
       equality : g₁ ≡ g₂
       equality = tran (sym (colim-splits-lim g₁)) (tran step-7 (colim-splits-lim g₂))
-}

--------------------------------------------------------------------------------


{-# OPTIONS --omega-in-omega --no-pattern-matching #-}

module IST.Results.MainTheorem where

open import IST.Base
open import IST.Util
open import IST.Approximation
open import IST.MetricSpaces
open import IST.Reals
open import IST.Naturals
open import IST.PredicatedTopologies
open import IST.Results.ExtensionTheorem
open import IST.Groups
open import IST.GroupActions
open import IST.NewmansTheorem


-- Theorem. Assume that the finite group H approximates the standard group G as a group via an
external
--          predicate ι. Consider a faithful K-Lipschitz faithful action of H on M, for some
standard K > 0.
--          Every periodic subgroup of $G$ also admits a standard faithful $K$-Lipschitz action
on $M$.
record MainTheorem : ESet where
  field
    G# : Group
    st-G# : st G#
    H# : FiniteGroup
```

```
    ι# : FiniteGroupApproximation H# G#
    M# : NewmanSpace
    st-M# : st M#
    A# : DiscreteAction H# (NewmanSpace.asMetricSpace M#)
  -- We first name everything in context.
  open Group G# renaming
    ( Carrier to G
    ; identity to 1G
    ; operation to xG
    ; inverse to iG
    ; assoc to G-associative
    ; unit-left to G-unit-left
    ; unit-right to G-unit-right
    ; inverse-left to G-inverse-left
    ; inverse-right to G-inverse-right
    )
  open FiniteGroup H# renaming
    ( Carrier to H
    ; identity to 1H
    ; operation to xH
    ; inverse to iH
    ; assoc to H-associative
    ; unit-left to H-unit-left
    ; unit-right to H-unit-right
    ; inverse-left to H-inverse-left
    ; inverse-right to H-inverse-right
    )
  open MetricSpace (NewmanSpace.asMetricSpace M#) renaming (Carrier to M)
  open DiscreteAction A# renaming (Map to act)
  open FiniteGroupApproximation ι# renaming (Map to ι)
  private
    M#' : MetricSpace
    M#' = NewmanSpace.asMetricSpace M#
    st-M#' : st M#'
    st-M#' = st-fun _ _ NewmanSpace.asMetricSpace M# st-NewmanSpace-asMetricSpace st-M#
    st-G : st G
    st-G = st-fun _ _ Group.Carrier G# st-Group-Carrier st-G#
    st-xG : st xG
    st-xG = st-fun-d _ _ Group.operation G# st-Group-operation st-G#
    st-1G : st 1G
    st-1G = st-fun-d _ _ Group.identity G# st-Group-identity st-G#
    st-M : st M
    st-M = st-fun _ _ MetricSpace.Carrier M#' st-MetricSpace-Carrier st-M#'
    st-distance : st distance
    st-distance = st-fun-d _ _ MetricSpace.distance M#' st-MetricSpace-distance st-M#'
    st-asMetricSpace-M# : st (NewmanSpace.asMetricSpace M#)
    st-asMetricSpace-M# =
      st-fun _ _ NewmanSpace.asMetricSpace M# st-NewmanSpace-asMetricSpace st-M#
    M## : HausdorffEquivalenceSpace
    M## = metric-to-hausdorff-equivalence (NewmanSpace.asMetricSpace M#) st-asMetricSpace-M#
  open HausdorffEquivalenceSpace M## renaming (Carrier to M-Carrier)
  -- The theorem requires one additional assumption to ensure the continuity of the resulting
  -- action. This can e.g. be a Lipschitz constant.
  field
    act-faithful : ∀ (g : H) → (g ≡ 1H → ⊥) → ∃ λ (m : M) → act g m ≡ m → ⊥
    isCompactSpace : IsCompactSpace M nearby
    K : ℝ
    st-K : st K
    positive-K : 0r < K
    lipschitz : ∀ (g : H) → ∀ (x y : M) → distance (act g x) (act g y) ≤ᵣ (K · distance x y)
  open IsCompactSpace isCompactSpace
  private
    K' : ℝ
    K' = inv K (λ _ → positive-K)
    st-K' : st K'
    st-K' = st-inv-v K (λ _ → positive-K) st-K
    positive-K' : 0r < K'
    positive-K' = <-inverse positive-K

    -- We prove the continuity of the action of H.
    S-continuity : ∀ (g : H) → ∀ (x : M) → st x → ∀ (y : M) → nearby x y → nearby (act g x) (act
g y)
    S-continuity g x st-x y x-near-y ε st-ε positive-ε = agx-near-agy where
      s : ℝ
      s = K' · ε
      st-s : st s
```

```
      st-s = st-fun _ _ (_·_ K') ε (st-fun _ _ _·_ K' st-· st-K') st-ε
      positive-s : 0r < s
      positive-s = step-3 where
        step-1 : K' · 0r < s
        step-1 = <-mult 0r ε K' positive-K' positive-ε
        step-2 : K' · 0r ≡ 0r
        step-2 = ·-null-left
        step-3 : 0r < s
        step-3 = transport step-2 {λ x → x < s} step-1
      dxy-under-s : distance x y < s
      dxy-under-s = x-near-y s st-s positive-s
      kdxy-under-ks : K · distance x y < K · s
      kdxy-under-ks = <-mult (distance x y) s K positive-K (x-near-y s st-s positive-s)
      kK'ε-equals-ε : K · s ≡ ε
      kK'ε-equals-ε = tran (tran step-1 step-2) step-3 where
        step-1 : K · (K' · ε) ≡ (K · K') · ε
        step-1 = sym ·-assoc
        step-2 : (K · K') · ε ≡ 1r · ε
        step-2 = cong (λ x → x · ε) (·-inverse-right (λ _ → positive-K))
        step-3 : 1r · ε ≡ ε
        step-3 = ·-unit-left
      kdxy-under-ε : K · distance x y < ε
      kdxy-under-ε = transport kK'ε-equals-ε {λ p → (K · distance x y) < p} kdxy-under-ks
      agx-near-agy : distance (act g x) (act g y) < ε
      agx-near-agy = by-cases _ case-1 case-2 (lipschitz g x y) where
        case-1 :  distance (act g x) (act g y) ≡ K · distance x y →
                  distance (act g x) (act g y) < ε
        case-1 p = transport (sym p) {λ p → p < ε} kdxy-under-ε
        case-2 : distance (act g x) (act g y) < K · distance x y →
                 distance (act g x) (act g y) < ε
        case-2 p = <-tran _ _ _ p kdxy-under-ε

    -- We prove that continuity of the action over a compact manifold implies uniform
continuity.
    -- TODO: move this proof to a more appropriate module.
    S-uniform-continuity : ∀ (g : H) → ∀ (x : M) → ∀ (y : M) → nearby x y → nearby (act g x)
(act g y)
    S-uniform-continuity g x y x-near-y = fx-near-fy where
      x' : M
      x' = proj₁ (compact x)
      st-x' : st x'
      st-x' = proj₁ (proj₂ (compact x))
      x'-near-x : nearby x' x
      x'-near-x = proj₂ (proj₂ (compact x))
      x'-near-y : nearby x' y
      x'-near-y = transitive _ _ _ x'-near-x x-near-y
      fx'-near-fx : nearby (act g x') (act g x)
      fx'-near-fx = S-continuity g x' st-x' x x'-near-x
      fx'-near-fy : nearby (act g x') (act g y)
      fx'-near-fy = S-continuity g x' st-x' y x'-near-y
      fx-near-fy : nearby (act g x) (act g y)
      fx-near-fy = transitive _ _ _ (symmetric _ _ fx'-near-fx) fx'-near-fy

    -- Group approximations of standard groups preserve and reflect unit elements.
    ι-preserves-unit : ι 1H 1G
    ι-preserves-unit = Map-preserves-unit st-G#

    ι-preserves-unit-Target : ∀ (g : G) → st g → ι 1H g → g ≡ 1G
    ι-preserves-unit-Target = Map-preserves-unit-Target st-G#

    -- We wish to apply the Extension Theorem to extend the action.
    -- To do that, we prove that a group approximation between H and G
    -- induces an appropriate set approximation between products (H × M)
    -- and (G × M).
    ι' : (H ∧ M) → G ∧ M → ESet
    ι' hm₁ gm₂ = ι (proj₁ hm₁) (proj₁ gm₂) *∧* internal (proj₂ hm₁ ≡ proj₂ gm₂)
    -- ι' (h , m₁) (g , m₂) = ι h g ∧* internal (m₁ ≡ m₂)

    ι'-exists : ∀ (gm : G ∧ M) → st gm → ∃* λ (hm : H ∧ M) → ι' hm gm
    ι'-exists gm st-gm = (h , m) , ι'-hm-gm where
      g : G
      g = proj₁ gm
      m : M
      m = proj₂ gm
      st-g : st g
```

```
  st-g = lemma-proj₁ (g , m) st-gm
  st-m : st m
  st-m = lemma-proj₂ (g , m) st-gm
  h : H
  h = proj₁ (Map-exists g st-g)
  ι-h-g : ι h g
  ι-h-g = proj₂ (Map-exists g st-g)
  ι'-hm-gm : ι' (h , m) (g , m)
  ι'-hm-gm = ι-h-g , fromInternal refl

ι'-unique-Source : ∀ (gm : G ∧ M) → st gm →
                   ∀ (h₁m : H ∧ M) → ι' h₁m gm →
                   ∀ (h₂m : H ∧ M) → ι' h₂m gm →
                   h₁m ≡ h₂m
ι'-unique-Source gm st-gm h₁m ι-h₁m-gm h₂m ι-h₂m-gm =
  h₁m-equals-h₂m where
  g : G
  g = proj₁ gm
  m : M
  m = proj₂ gm
  h₁ : H
  h₁ = proj₁ h₁m
  m₁ : M
  m₁ = proj₂ h₁m
  h₂ : H
  h₂ = proj₁ h₂m
  m₂ : M
  m₂ = proj₂ h₂m
  st-g : st g
  st-g = lemma-proj₁ (g , m) st-gm
  st-m : st m
  st-m = lemma-proj₂ (g , m) st-gm
  m₁-equals-m : m₁ ≡ m
  m₁-equals-m = toInternal _ (proj₂ ι-h₁m-gm)
  m₂-equals-m : m₂ ≡ m
  m₂-equals-m = toInternal _ (proj₂ ι-h₂m-gm)
  m₁-equals-m₂ : m₁ ≡ m₂
  m₁-equals-m₂ = tran m₁-equals-m (sym m₂-equals-m)
  h₁-equals-h₂ : h₁ ≡ h₂
  h₁-equals-h₂ = Map-unique-Source g st-g h₁ (proj₁ ι-h₁m-gm) h₂ (proj₁ ι-h₂m-gm)
  pair : H → M → H ∧ M
  pair x y = (x , y)
  product-lemma : ∀ {x₁ x₂ : H} → ∀ {y₁ y₂ : M} →
                  x₁ ≡ x₂ → y₁ ≡ y₂ → (pair x₁ y₁) ≡ (pair x₂ y₂)
  product-lemma = lemma-product-equality
  h₁m-equals-h₂m : (h₁ , m₁) ≡ (h₂ , m₂)
  h₁m-equals-h₂m = product-lemma h₁-equals-h₂ m₁-equals-m₂

ι'-unique-Target : ∀ (g₁m : G ∧ M) → st g₁m →
                   ∀ (g₂m : G ∧ M) → st g₂m →
                   ∀ (hm : H ∧ M) → ι' hm g₁m → ι' hm g₂m →
                   g₁m ≡ g₂m
ι'-unique-Target g₁m st-g₁m g₂m st-g₂m hm ι'-hm-g₁m ι'-hm-g₂m =
  g₁m-equals-g₂m where
  g₁ : G
  g₁ = proj₁ g₁m
  m₁ : M
  m₁ = proj₂ g₁m
  g₂ : G
  g₂ = proj₁ g₂m
  m₂ : M
  m₂ = proj₂ g₂m
  h : H
  h = proj₁ hm
  m : M
  m = proj₂ hm
  st-g₁ : st g₁
  st-g₁ = lemma-proj₁ (g₁ , m₁) st-g₁m
  st-g₂ : st g₂
  st-g₂ = lemma-proj₁ (g₂ , m₂) st-g₂m
  st-m₁ : st m₁
```

```
      st-m₁ = lemma-proj₂ (g₁ , m₁) st-g₁m
      st-m₂ : st m₂
      st-m₂ = lemma-proj₂ (g₂ , m₂) st-g₂m
      m₁-equals-m : m ≡ m₁
      m₁-equals-m = toInternal _ (proj₂ ι'-hm-g₁m)
      m₂-equals-m : m ≡ m₂
      m₂-equals-m = toInternal _ (proj₂ ι'-hm-g₂m)
      m₁-equals-m₂ : m₁ ≡ m₂
      m₁-equals-m₂ = tran (sym m₁-equals-m) (m₂-equals-m)
      ι-h-g₁ : ι h g₁
      ι-h-g₁ = proj₁ ι'-hm-g₁m
      ι-h-g₂ : ι h g₂
      ι-h-g₂ = proj₁ ι'-hm-g₂m
      g₁-equals-g₂ : g₁ ≡ g₂
      g₁-equals-g₂ = Map-unique-Target g₁ st-g₁ g₂ st-g₂ h ι-h-g₁ ι-h-g₂
      pair : G → M → G ∧ M
      pair x y = (x , y)
      product-lemma : ∀ {x₁ x₂ : G} → ∀ {y₁ y₂ : M} →
                     x₁ ≡ x₂ → y₁ ≡ y₂ → (pair x₁ y₁) ≡ (pair x₂ y₂)
      product-lemma = lemma-product-equality
      g₁m-equals-g₂m : (g₁ , m₁) ≡ (g₂ , m₂)
      g₁m-equals-g₂m = product-lemma g₁-equals-g₂ m₁-equals-m₂

    st-G∧M : st (G ∧ M)
    st-G∧M = st-fun _ _ (_∧_ G) M (st-fun _ _ _∧_ G st-∧ st-G) st-M

    A#∧M : Approximation (H ∧ M) (G ∧ M)
    A#∧M = record { Map = ι'
                  ; isApproximation = record { Target-st = st-G∧M
                                             ; Map-exists = ι'-exists
                                             ; Map-unique-Source = ι'-unique-Source
                                             ; Map-unique-Target = ι'-unique-Target
                                             }
                  }
    -- Now we can invoke the extension theorem to extend the action to a map G × M → M.
    by-extension : ExtensionTheorem
    by-extension =
      record { G = G ∧ M
             ; H = H ∧ M
             ; A = A#∧M
             ; M# = record
                      { Carrier = M
                      ; nearby = nearby
                      ; isHausdorffSpace = isHausdorffSpace
                      ; isCompactSpace = isCompactSpace
                      }
             ; st-M = st-M
             ; f = λ hm → act (proj₁ hm) (proj₂ hm) }
    open ExtensionTheorem by-extension hiding (G; H; A; M#; st-M; f) renaming
      ( f' to act-G
      ; st-f' to st-act-G
      ; f'-exists to act-G-exists
      ; f'-exists-st to act-G-exists-st
      ; f'-unique to act-G-unique
      )

    -- The extension theorem extends the action with signature H × M → M to a
    -- function with signature G × M → M. Here we prove the result standard-valued.
    act' : G → M → M
    act' g m = proj₁ (act-G-exists (g , m))

    act'-property : ∀ (g : G) → ∀ (m : M) → act-G ((g , m) , act' g m)
    act'-property g m = proj₂ (act-G-exists (g , m))

    act'-st-valued : ∀ (g : G) → st g → ∀ (m : M) → st m → st (act' g m)
    act'-st-valued g st-g m st-m = st-act'-g-m where
      gm : G ∧ M
      gm = (g , m)
      sm : ∃* λ (m' : M) → st m' *∧* internal (act-G ((g , m) , m'))
      sm = act-G-exists-st (g , m) (lemma-pairing g m st-g st-m)
      st-sm : st (proj₁ sm)
      st-sm = proj₁ (proj₂ sm)
      f'-gm-sm : act-G (gm , (proj₁ sm))
```

```
      f'-gm-sm = toInternal _ (proj₂ (proj₂ sm))
      sm-equals-act-G-m : proj₁ sm ≡ act' g m -- act-G ((g , m) , ?)
      sm-equals-act-G-m = act-G-unique (g , m) (proj₁ sm) (act' g m) f'-gm-sm (act'-property g
m)
      st-act'-g-m : st (act' g m)
      st-act'-g-m = transport* sm-equals-act-G-m {st} st-sm

    act'-property-st : ∀ (g : G) → st g → ∀ (m : M) → st m →
                       ∃* λ (hm : H ∧ M) → ι' hm (g , m) *∧* nearby (act' g m) (act (proj₁ hm)
(proj₂ hm))
    act'-property-st g st-g m st-m =
      ax-Standard-2 _ ((g , m) , act' g m) act-G-pair (act'-property g m) where
      st-gm : st (g , m)
      st-gm = lemma-pairing g m st-g st-m
      act-G-pair : st ((g , m) , act' g m)
      act-G-pair = lemma-pairing (g , m) (act' g m) st-gm (act'-st-valued g st-g m st-m)

    -- The main lemma: if h approximates g, then the result of the action of h
    -- lies near the result of the action of g.
    act'-lemma : ∀ (g : G) → st g → ∀ (m : M) → st m → ∀ (h : H) → ι h g →
                 nearby (act' g m) (act h m)
    act'-lemma g st-g m st-m h ι-h-g = agm-near-ahm where
      ι'-hm-gm : ι' (h , m) (g , m)
      ι'-hm-gm = ι-h-g , fromInternal refl
      hm'-exists : ∃* λ (hm' : H ∧ M) → ι' hm' (g , m) *∧* nearby (act' g m) (act (proj₁ hm')
(proj₂ hm'))
      hm'-exists = act'-property-st g st-g m st-m
      h' : H
      h' = proj₁ (proj₁ hm'-exists)
      m' : M
      m' = proj₂ (proj₁ hm'-exists)
      ι'-hm'-gm : ι' (h' , m') (g , m)
      ι'-hm'-gm = proj₁ (proj₂ hm'-exists)
      hm'-equals-hm : (h' , m') ≡ (h , m)
      hm'-equals-hm =
        ι'-unique-Source (g , m) (lemma-pairing g m st-g st-m) (h' , m') ι'-hm'-gm (h , m) ι'-
hm-gm
      h'-equals-h : h' ≡ h
      h'-equals-h = cong proj₁ hm'-equals-hm
      m'-equals-m : m' ≡ m
      m'-equals-m = cong proj₂ hm'-equals-hm
      agm-near-ahm' : nearby (act' g m) (act h' m')
      agm-near-ahm' = proj₂ (proj₂ hm'-exists)
      agm-near-ahm : nearby (act' g m) (act h m)
      agm-near-ahm =
        transport* h'-equals-h {λ z → nearby (act' g m) (act z m)}
          (transport* m'-equals-m {λ z → nearby (act' g m) (act h' z)} agm-near-ahm')

    -- First we prove that the identity acts via the identity function.
    act'-identity-st : ∀ (m : M) → st m → internal (act' 1G m ≡ m)
    act'-identity-st m st-m = fromInternal a1Gm-equals-m where
      a1Gm-near-a1Hm : nearby (act' 1G m) (act 1H m)
      a1Gm-near-a1Hm = act'-lemma 1G st-1G m st-m 1H ι-preserves-unit
      a1Gm-near-m : nearby (act' 1G m) m
      a1Gm-near-m = transport* (action-identity m) {λ z → nearby (act' 1G m) z} a1Gm-near-a1Hm
      a1Gm-equals-m : act' 1G m ≡ m
      a1Gm-equals-m = hausdorff (act' 1G m) st-a1Gm m st-m m a1Gm-near-m (reflexive m) where
        st-a1Gm : st (act' 1G m)
        st-a1Gm = act'-st-valued 1G st-1G m st-m

    act'-identity : ∀ (m : M) → act' 1G m ≡ m
    act'-identity = ax-Transfer-EI (∀' M (λ m → int' (act' 1G m ≡ m))) act'-identity-st std-Φ
where
      Φ : TransferPred
      Φ = ∀' M λ m → int' (act' 1G m ≡ m)
      std-Φ : st M *∧* ∀ (m : M) → st m → st (act' 1G m ≡ m)
      std-Φ = st-M , λ m st-m →
              st-fun _ _ (eq (act' 1G m)) m
              (st-fun _ _ eq (act' 1G m) st-eq (help1 m st-m)) st-m where
        eq : M → M → Set
        eq = _≡_
        st-eq : st eq
        st-eq = st-≡-full
        help1 : (m : M) → st m → st (act' 1G m)
```

```
        help1 m st-m = act'-st-valued 1G st-1G m st-m

    -- Now we prove that the action is a homomorphism with respect to the operations.
    act'-operation-st : ∀ (g : G) → st g → ∀ (h : G) → st h → ∀ (m : M) → st m →
                        internal (act' g (act' h m) ≡ act' (xG g h) m)
    act'-operation-st g st-g h st-h m st-m = fromInternal (sym (a'ghm-equals-a'ga'hm)) where
      -- Book-keeping: We must prove that if g' approximates g and
      -- h' approximates h then g'h' approximates gh.
      gh : G
      gh = xG g h
      st-gh : st gh
      st-gh = st-fun _ _ (xG g) h (st-fun _ _ xG g st-xG st-g) st-h
      g' : H
      g' = proj₁ (Map-exists g st-g)
      ι-g'-g : ι g' g
      ι-g'-g = proj₂ (Map-exists g st-g)
      h' : H
      h' = proj₁ (Map-exists h st-h)
      ι-h'-h : ι h' h
      ι-h'-h = proj₂ (Map-exists h st-h)
      g'h' : H
      g'h' = xH g' h'
      ι-g'h'-gh : ι g'h' gh
      ι-g'h'-gh = Map-homomorphism g' h' g st-g h st-h ι-g'-g ι-h'-h
      -- It follows on one hand that applying gh to m results in a neighbor
      -- of applying g'h' to m.
      a'ghm-near-ag'ah'm : nearby (act' gh m) (act g'h' m)
      a'ghm-near-ag'ah'm = act'-lemma gh st-gh m st-m g'h' ι-g'h'-gh
      ag'h'm-equals-ag'ah'm : act g'h' m ≡ act g' (act h' m)
      ag'h'm-equals-ag'ah'm = sym (action-operation g' h' m)
      one-hand : nearby (act' gh m) (act g' (act h' m))
      one-hand = transport* ag'h'm-equals-ag'ah'm {λ z → nearby (act' gh m) z} a'ghm-near-
ag'ah'm
      -- It follows on the other hand that the result of applying g' to h' at m
      -- neighbors the same element.
      a'ga'hm-near-ag'a'hm : nearby (act' g (act' h m)) (act g' (act' h m))
      a'ga'hm-near-ag'a'hm = act'-lemma g st-g (act' h m) (act'-st-valued h st-h m st-m) g' ι-
g'-g
      a'hm-near-ah'm : nearby (act' h m) (act h' m)
      a'hm-near-ah'm = act'-lemma h st-h m st-m h' ι-h'-h
      ag'a'hm-near-ag'ah'm : nearby (act g' (act' h m)) (act g' (act h' m))
      ag'a'hm-near-ag'ah'm = S-uniform-continuity g' (act' h m) (act h' m) a'hm-near-ah'm
      other-hand : nearby (act' g (act' h m)) (act g' (act h' m))
      other-hand = transitive _ _ _ a'ga'hm-near-ag'a'hm ag'a'hm-near-ag'ah'm
      -- These both satisfy standardness!
      st-one : st (act' gh m)
      st-one = act'-st-valued gh st-gh m st-m
      st-other : st (act' g (act' h m))
      st-other = act'-st-valued g st-g (act' h m) (act'-st-valued h st-h m st-m)
      -- By Hausdorff separation standard values with common neighbors are equal.
      a'ghm-equals-a'ga'hm : act' gh m ≡ act' g (act' h m)
      a'ghm-equals-a'ga'hm =
        hausdorff (act' gh m) st-one
                  (act' g (act' h m)) st-other
                  (act g' (act h' m)) one-hand other-hand

    act'-operation : ∀ (g : G) → ∀ (h : G) → ∀ (m : M) → act' g (act' h m) ≡ act' (xG g h) m
    act'-operation = ax-Transfer-EI Φ act'-operation-st std-Φ where
      Φ : TransferPred
      Φ = ∀' G λ g → ∀' G λ h → ∀' M λ m → int' (act' g (act' h m) ≡ act' (xG g h) m)
      eq : M → M → Set
      eq = _≡_
      st-eq : st eq
      st-eq = st-≡-full
      st-one : ∀ (g h : G) → ∀ (m : M) → st g → st h → st m → st (act' (xG g h) m)
      st-one g h m st-g st-h st-m =
        act'-st-valued (xG g h) (st-fun _ _ (xG g) h (st-fun _ _ xG g st-xG st-g) st-h) m st-m
      st-other : ∀ (g h : G) → ∀ (m : M) → st g → st h → st m → st (act' g (act' h m))
      st-other g h m st-g st-h st-m = act'-st-valued g st-g (act' h m) (act'-st-valued h st-h m
st-m)
      std-Φ : st G *∧* ∀ (g : G) → st g → st G *∧* ∀ (h : G) → st h → st M *∧*
              ∀ (m : M) → st m → st (act' g (act' h m) ≡ act' (xG g h) m)
      std-Φ = st-G , λ g st-g →
              st-G , λ h st-h →
              st-M , λ m st-m →  st-fun _ _ (eq (act' g (act' h m))) (act' (xG g h) m)
```

```
              (st-fun _ _ eq (act' g (act' h m)) st-eq (st-other g h m st-g st-h st-m))
              (st-one g h m st-g st-h st-m)
    -- At this point we know that act' has all the properties of an action of G on M.
    -- But it might be a trivial action - we have to rule that out!

    -- Before discussing faithfulness, we note that act' is a standard action, and therefore
    -- it satisfies both S-continuity and ε-δ continuity.

    act'-lipschitz-st! :  ∀ (g : G) → st g →
                          ∀ (x : M) → st x →
                          ∀ (y : M) → st y → internal (
                            distance (act' g x) (act' g y) ≤ᵣ (K · distance x y)
                          )
    act'-lipschitz-st! g st-g x st-x y st-y = fromInternal difference-0 where
      h : H
      h = proj₁ (Map-exists g st-g)
      ι-h-g : ι h g
      ι-h-g = proj₂ (Map-exists g st-g)
      difference-ε-st : ∀ (ε : ℝ) → st ε → 0r < ε → distance (act' g x) (act' g y) ≤ᵣ K · distance x y + ε
      difference-ε-st ε st-ε positive-ε = by-cases _ case-1 case-2 (lipschitz h y x) where
        ε/2 : ℝ
        ε/2 = ε /2r
        st-ε/2 : st ε/2
        st-ε/2 = st-/2r-v ε st-ε
        positive-ε/2 : 0r < ε/2
        positive-ε/2 = pos-/2r-v ε positive-ε
        case-2 : distance (act h y) (act h x) < K · distance y x →
                 distance (act' g x) (act' g y) ≤ᵣ K · distance x y + ε
        case-2 final-1 = inr final-13 where
          final-2 : distance (act h y) (act h x) < K · distance x y
          final-2 = transport (symmetry y x) {λ z → distance (act h y) (act h x) < K · z} final-1
          final-3 : distance (act' g x) (act h x) < ε/2
          final-3 = act'-lemma g st-g x st-x h ι-h-g ε/2 st-ε/2 positive-ε/2
          final-4 : distance (act h x) (act' g x) < ε/2
          final-4 = transport (symmetry (act' g x) (act h x)) {λ z → z < ε/2} final-3
          final-5 : distance (act h y) (act h x) + distance (act h x) (act' g x) < K · distance x y + ε/2
          final-5 = <-plus-both (distance (act h y) (act h x)) _ _ _ final-2 final-4
          final-6 : distance (act h y) (act' g x) < K · distance x y + ε/2
          final-6 = triangle (act h y) (act h x) (act' g x) (K · distance x y + ε/2) final-5
          final-7 : distance (act' g y) (act h y) < ε/2
          final-7 = act'-lemma g st-g y st-y h ι-h-g ε/2 st-ε/2 positive-ε/2
          final-8 : distance (act h y) (act' g y) < ε/2
          final-8 = transport (symmetry (act' g y) (act h y)) {λ z → z < ε/2} final-7
          final-9 : distance (act' g x) (act h y) < K · distance x y + ε/2
          final-9 = transport (symmetry (act h y) (act' g x)) {λ z → z < K · distance x y + ε/2} final-6
          final-10 :
            distance (act' g x) (act h y) + distance (act h y) (act' g y) < (K · distance x y + ε/2) + ε/2
          final-10 = <-plus-both (distance (act' g x) (act h y)) _ _ _ final-9 final-8
          final-11 : distance (act' g x) (act' g y) < (K · distance x y + ε/2) + ε/2
          final-11 = triangle (act' g x) (act h y) (act' g y)
                         ((K · distance x y + ε/2) + ε/2) final-10
          final-12 : distance (act' g x) (act' g y) < K · distance x y + ε/2 + ε/2
          final-12 = transport +-assoc {λ z → distance (act' g x) (act' g y) < z} final-11
          final-13 : distance (act' g x) (act' g y) < K · distance x y + ε
          final-13 =
            transport (/2r-half {ε}) {λ z → distance (act' g x) (act' g y) < K · distance x y + z} final-12

        case-1 : distance (act h y) (act h x) ≡ K · distance y x →
                 distance (act' g x) (act' g y) ≤ᵣ K · distance x y + ε
        case-1 final-1 = final-x12 where
          final-x1 :
            distance (act' g x) (act' g y) ≤ᵣ distance (act' g x) (act h y) + distance (act h y) (act' g y)
          final-x1 = triangle-≤ᵣ (act' g x) (act h y) (act' g y)
          final-x2 :
            distance (act h y) (act' g x) ≤ᵣ distance (act h y) (act h x) + distance (act h x) (act' g x)
          final-x2 = triangle-≤ᵣ (act h y) (act h x) (act' g x)
          final-x3 :
```

```
            distance (act' g x) (act' g y) ≤ᵣ distance (act h y) (act' g x) + distance (act h y)
(act' g y)
          final-x3 = transport (symmetry (act' g x) (act h y))
                       {λ p → distance (act' g x) (act' g y) ≤ᵣ p + distance (act h y) (act' g
y)} final-x1
          final-x4 :
             distance (act h y) (act' g x) + distance (act h y) (act' g y) ≤ᵣ
             (distance (act h y) (act h x) + distance (act h x) (act' g x)) + distance (act h y)
(act' g y)
          final-x4 = ≤ᵣ-plus _ _ (distance (act h y) (act' g y)) final-x2
          final-x5 :
             distance (act' g x) (act' g y) ≤ᵣ
             (distance (act h y) (act h x) + distance (act h x) (act' g x)) + distance (act h y)
(act' g y)
          final-x5 = ≤ᵣ-tran _ _ _ final-x3 final-x4
          final-x6 :
             distance (act' g x) (act' g y) ≤ᵣ
             distance (act h y) (act h x) + (distance (act h x) (act' g x) + distance (act h y)
(act' g y))
          final-x6 = transport +-assoc {λ p → distance (act' g x) (act' g y) ≤ᵣ p} final-x5
          final-3 : distance (act' g x) (act h x) < ε/2
          final-3 = act'-lemma g st-g x st-x h ι-h-g ε/2 st-ε/2 positive-ε/2
          final-4 : distance (act h x) (act' g x) < ε/2
          final-4 = transport (symmetry (act' g x) (act h x)) {λ z → z < ε/2} final-3
          final-7 : distance (act' g y) (act h y) < ε/2
          final-7 = act'-lemma g st-g y st-y h ι-h-g ε/2 st-ε/2 positive-ε/2
          final-8 : distance (act h y) (act' g y) < ε/2
          final-8 = transport (symmetry (act' g y) (act h y)) {λ z → z < ε/2} final-7
          final-x7 :
            distance (act h x) (act' g x) + distance (act h y) (act' g y) ≤ᵣ ε/2 + ε/2
          final-x7 = ≤ᵣ-plus-both _ _ _ _ (inr final-4) (inr final-8)
          final-x8 :
            distance (act h x) (act' g x) + distance (act h y) (act' g y) ≤ᵣ ε
          final-x8 = transport (/2r-half {ε})
                       {λ p → distance (act h x) (act' g x) + distance (act h y) (act' g y) ≤ᵣ
p}
                       final-x7
          final-x9 :
            distance (act h y) (act h x) + (distance (act h x) (act' g x) + distance (act h y)
(act' g y)) ≤ᵣ
            distance (act h y) (act h x) + ε
          final-x9 = ≤ᵣ-plus-left _ _ (distance (act h y) (act h x)) final-x8
          final-x10 :
            distance (act' g x) (act' g y) ≤ᵣ distance (act h y) (act h x) + ε
          final-x10 = ≤ᵣ-tran _ _ _ final-x6 final-x9
          final-x11 :
            distance (act' g x) (act' g y) ≤ᵣ K · distance y x + ε
          final-x11 = transport final-1 {λ p → distance (act' g x) (act' g y) ≤ᵣ p + ε} final-
x10
          final-x12 :
            distance (act' g x) (act' g y) ≤ᵣ K · distance x y + ε
          final-x12 = transport (symmetry y x) {λ p → distance (act' g x) (act' g y) ≤ᵣ K · p +
ε} final-x11

      difference-ε : ∀ (ε : ℝ) → 0r < ε → distance (act' g x) (act' g y) ≤ᵣ K · distance x y +
ε
      difference-ε = ax-Transfer-EI Φ (λ ε → λ st-ε → fromInternal (difference-ε-st ε st-ε))
std-Φ where
        Φ : TransferPred
        Φ = ∀' ℝ λ ε → int' (0r < ε → distance (act' g x) (act' g y) ≤ᵣ K · distance x y + ε)
        std-Φ :
          st ℝ *∧* (∀ (a : ℝ) → st a → st (0r < a → distance (act' g x) (act' g y) ≤ᵣ K ·
distance x y + a))
        std-Φ =
          st-ℝ , λ a st-a →
          st-→ _ (st-fun _ _ (_<_ 0r) a (st-fun _ _ _<_ 0r st-< st-0r) st-a) _
             (st-fun _ _ (_≤ᵣ_ (distance (act' g x) (act' g y)))
             (K · distance x y + a)
             (st-fun _ _ _≤ᵣ_ (distance (act' g x) (act' g y))
             st-≤ᵣ (st-fun _ _ (distance (act' g x)) (act' g y)
             (st-fun _ _ distance (act' g x)
             st-distance (act'-st-valued g st-g x st-x))
             (act'-st-valued g st-g y st-y)))
             (st-fun _ _ (_+_ (K · distance x y)) a
             (st-fun _ _ _+_ (K · distance x y)
             st-+ (st-fun _ _ (_·_ K) (distance x y)
```

```
              (st-fun _ _ _·_ K st-· st-K)
              (st-fun _ _ (distance x) y (st-fun _ _ distance x st-distance st-x) st-y))) st-a))
      difference-0 : distance (act' g x) (act' g y) ≤ᵣ K · distance x y
      difference-0 = lemma-ε-of-room-plus-≤ᵣ _ _ difference-ε

    act'-lipschitz! :  ∀ (g : G) →
                       ∀ (x : M) →
                       ∀ (y : M) →
                       distance (act' g x) (act' g y) ≤ᵣ (K · distance x y)
    act'-lipschitz! = ax-Transfer-EI Φ act'-lipschitz-st! std-Φ where
      Φ : TransferPred
      Φ = ∀' G λ g → ∀' M λ x → ∀' M λ y → int' (distance (act' g x) (act' g y) ≤ᵣ K · distance
x y)
      std-Φ : st G *∧* (∀ (g : G) → st g → st M *∧* (∀ (x : M) → st x → st M *∧* (∀ (y : M) → st
y →
              st (distance (act' g x) (act' g y) ≤ᵣ K · distance x y))))
      std-Φ = st-G , λ g st-g → st-M , λ x st-x → st-M , λ y st-y →
        st-fun _ _ (_≤ᵣ_ (distance (act' g x) (act' g y)))
          (K · distance x y)
          (st-fun _ _ _≤ᵣ_ (distance (act' g x) (act' g y))
          st-≤ᵣ (st-fun _ _ (distance (act' g x)) (act' g y)
          (st-fun _ _ distance (act' g x)
          st-distance (act'-st-valued g st-g x st-x)) (act'-st-valued g st-g y st-y)))
          (st-fun _ _ (_·_ K) (distance x y)
          (st-fun _ _ _·_ K st-· st-K) (st-fun _ _ (distance x) y
          (st-fun _ _ distance x st-distance st-x) st-y))

    act'-S-uniform-continuity : ∀ (g : G) →
                                ∀ (x : M) →
                                ∀ (y : M) → nearby x y → nearby (act' g x) (act' g y)
    act'-S-uniform-continuity g x y x-near-y ε st-ε positive-ε = agx-near-agy where
      s : ℝ
      s = K' · ε
      st-s : st s
      st-s = st-fun _ _ (_·_ K') ε (st-fun _ _ _·_ K' st-· st-K') st-ε
      positive-s : 0r < s
      positive-s = step-3 where
        step-1 : K' · 0r < s
        step-1 = <-mult 0r ε K' positive-K' positive-ε
        step-2 : K' · 0r ≡ 0r
        step-2 = ·-null-left
        step-3 : 0r < s
        step-3 = transport step-2 {λ x → x < s} step-1
      dxy-under-s : distance x y < s
      dxy-under-s = x-near-y s st-s positive-s
      kdxy-under-ks : K · distance x y < K · s
      kdxy-under-ks = <-mult (distance x y) s K positive-K (x-near-y s st-s positive-s)
      kK'ε-equals-ε : K · s ≡ ε
      kK'ε-equals-ε = tran (tran step-1 step-2) step-3 where
        step-1 : K · (K' · ε) ≡ (K · K') · ε
        step-1 = sym ·-assoc
        step-2 : (K · K') · ε ≡ 1r · ε
        step-2 = cong (λ x → x · ε) (·-inverse-right (λ _ → positive-K))
        step-3 : 1r · ε ≡ ε
        step-3 = ·-unit-left
      kdxy-under-ε : K · distance x y < ε
      kdxy-under-ε = transport kK'ε-equals-ε {λ p → (K · distance x y) < p} kdxy-under-ks
      agx-near-agy : distance (act' g x) (act' g y) < ε
      agx-near-agy = by-cases _ case-1 case-2 (act'-lipschitz! g x y) where
        case-1 : distance (act' g x) (act' g y) ≡ K · distance x y →
                 distance (act' g x) (act' g y) < ε
        case-1 p = transport (sym p) {λ p → p < ε} kdxy-under-ε
        case-2 : distance (act' g x) (act' g y) < K · distance x y →
                 distance (act' g x) (act' g y) < ε
        case-2 p = <-tran _ _ _ p kdxy-under-ε

    act'-continuity : ∀ (g : G) → ∀ (m : M) →
                      ∀ (ε : ℝ) → 0r < ε →
                      ∃ λ (δ : ℝ) → (0r < δ) ∧ (
                      ∀ (m' : M) → distance m m' < δ → distance (act' g m) (act' g m') < ε)
    act'-continuity x m ε positive-ε = K' · ε , (positive-K'ε , helper) where
      positive-K'ε : 0r < K' · ε
      positive-K'ε = transport (·-null-left) {λ z → z < K' · ε} (<-mult 0r ε K' positive-K'
positive-ε)
      helper : (m' : M) → distance m m' < K' · ε → distance (act' x m) (act' x m') < ε
```

```
    helper m' p = step-5 where
      step-1 : distance (act' x m) (act' x m') ≤ᵣ K · distance m m'
      step-1 = act'-lipschitz! x m m'
      step-2 : K · distance m m' < K · (K' · ε)
      step-2 = <-mult _ _ _ positive-K p
      step-3 : K · (K' · ε) ≡ ε
      step-3 =
        tran (sym (·-assoc {K} {K'} {ε})) (
        tran (cong (λ z → z · ε) (
        ·-inverse-right (λ _ → positive-K))) ·-unit-left)
      step-4 : K · distance m m' < ε
      step-4 = transport step-3 {λ z → K · distance m m' < z} step-2
      step-5 : distance (act' x m) (act' x m') < ε
      step-5 = by-cases _ case-1 case-2 step-1 where
        case-1 : distance (act' x m) (act' x m') ≡ K · distance m m' →
                 distance (act' x m) (act' x m') < ε
        case-1 p = transport (sym p) {λ p → p < ε} step-4
        case-2 : distance (act' x m) (act' x m') < K · distance m m' →
                 distance (act' x m) (act' x m') < ε
        case-2 p = <-tran _ _ _ p step-4

-- We prove that the action G × M → M we constructed satisfies faithfulness on every
-- finite subgroup X < G. We need ∀X.∀x∈X.∃m∈M. x≠1 → x@m≠m. By internality, it suffices
-- to prove ∀ˢᵗX.∀ˢᵗx∈X.∃m∈M. x≠1 → x@m≠m, so we establish the latter.
record Faithfulness : ESet where
  field
    X<G : PeriodicSubgroup G#
  open PeriodicSubgroup X<G renaming
    ( Source to X#
    ; Map to emb
    ; Map-identity to emb-identity
    ; Map-operation to emb-operation
    ; Map-injective to emb-injective
    ; Map-power to emb-power
    )
  field
    st-X# : st X#
    st-emb : st emb
  open PeriodicGroup X# renaming
    ( Carrier to X
    ; identity to 1X
    ; operation to xX
    ; inverse to iX
    ; assoc to X-associative
    ; unit-left to X-unit-left
    ; unit-right to X-unit-right
    ; inverse-left to X-inverse-left
    ; inverse-right to X-inverse-right
    ; order to X-order
    ; order-minimal to X-order-minimal
    )

  st-X : st X
  st-X = st-fun _ _ PeriodicGroup.Carrier X# st-PeriodicGroup-Carrier st-X#

  st-1X : st 1X
  st-1X = st-fun-d _ _ PeriodicGroup.identity X# st-PeriodicGroup-identity st-X#

  st-X-order : st X-order
  st-X-order = st-fun-d _ _ PeriodicGroup.order X# st-PeriodicGroup-order st-X#

  -- We prove that X acts on M using a meet-in-the-middle argument.
  xact : X → M → M
  xact x m = act' (emb x) m

  xact-st-valued : ∀ (x : X) → st x → ∀ (m : M) → st m → st (act' (emb x) m)
  xact-st-valued x st-x m st-m = act'-st-valued (emb x) (st-fun _ _ emb x st-emb st-x) m st-m

  xact-identity : ∀ (m : M) → xact 1X m ≡ m
  xact-identity m = tran xact-1X-equals-act'-1G (act'-identity m) where
    xact-1X-equals-act'-1G : xact 1X m ≡ act' 1G m
    xact-1X-equals-act'-1G = transport emb-identity {λ z → xact 1X m ≡ act' z m} refl

  xact-operation : ∀ (x y : X) (m : M) → xact x (xact y m) ≡ xact (xX x y) m
  xact-operation x y m = tran step-1 step-2 where
    step-1 : xact x (xact y m) ≡ act' (xG (emb x) (emb y)) m
```

```
      step-1 = act'-operation (emb x) (emb y) m
      step-2 : act' (xG (emb x) (emb y)) m ≡ act' (emb (xX x y)) m
      step-2 = cong (λ z → act' z m) (sym (emb-operation x y))

    xact-continuity : ∀ (x : X) → ∀ (m : M) →
                      ∀ (ε : ℝ) → 0r < ε →
                      ∃ λ (δ : ℝ) → (0r < δ) ∧ (
                      ∀ (m' : M) → distance m m' < δ → distance (xact x m) (xact x m') < ε)
    xact-continuity x m ε positive-ε = act'-continuity (emb x) m ε positive-ε

    X-Action : PeriodicDiscreteAction X# M#'
    X-Action =
      record { Map = xact
             ; isPeriodicDiscreteAction =
               record { isGroupAction =
                 record { action-identity = xact-identity
                        ; action-operation = xact-operation }
                      ; continuity = xact-continuity
                      }
             }

    module Given (x : X) (st-x : st x) (x-not-id : x ≡ 1X → ⊥) where
      st-emb-x : st (emb x)
      st-emb-x = st-fun _ _ emb x st-emb st-x

      -- We have a standard element x ∈ G, so we can pick a h with ι(h,x).
      h : H
      h = proj₁ (Map-exists (emb x) st-emb-x)

      ι-h-x : ι h (emb x)
      ι-h-x = proj₂ (Map-exists (emb x) st-emb-x)

      -- Since x≠1, h≠1.

      emb-x-not-id : emb x ≡ 1G → ⊥
      emb-x-not-id emb-x-equals-id = x-not-id (emb-injective _ _ (tran emb-x-equals-id (sym emb-
identity)))

      h-not-id : h ≡ 1H → ⊥
      h-not-id h-equals-id = emb-x-not-id step-2 where
        step-1 : ι 1H (emb x)
        step-1 = transport* h-equals-id {λ z → ι z (emb x)} ι-h-x
        step-2 : emb x ≡ 1G
        step-2 = Map-unique-Target (emb x) st-emb-x 1G st-1G 1H step-1 ι-preserves-unit

      -- We prove that ι(h,x) → ι(hⁿ,xⁿ) for all standard n ∈ ℕ. Note that this requires
      -- a style of argument known as external induction, and the implication would not
      -- hold for nonstandard n.

      ι-hn-xn : ∀ (n : ℕ) → st n → ι (FiniteGroup.power H# h n) (Group.power G# (emb x) n)
      ι-hn-xn n st-n = external-induction
                         {λ n → ι (FiniteGroup.power H# h n) (Group.power G# (emb x) n)}
                         (Map-preserves-unit st-G#) ψ-inductive n st-n where
        ψ-inductive : ∀ k → st k → ι (FiniteGroup.power H# h k) (Group.power G# (emb x) k) →
                      ι (FiniteGroup.power H# h (suc k)) (Group.power G# (emb x) (suc k))
        ψ-inductive k st-k ι-hk-xk =
          Map-homomorphism h hk (emb x) st-emb-x xk st-xk ι-h-x ι-hk-xk where
          hk : H
          hk = FiniteGroup.power H# h k
          xk : G
          xk = Group.power G# (emb x) k
          st-xk : st xk
          st-xk =
            st-fun _ _ (Group.power G# (emb x)) k
            (st-fun _ _ (Group.power G#) (emb x)
            (st-fun-d _ _ Group.power G# st-Group-power st-G#) st-emb-x) st-k

      -- Since we have ι(hⁿ,xⁿ) for all standard n∈ℕ, and the order ord(x) belongs to the
      -- standard naturals, it follows that ord(h) < ord(x), and hence ord(h) also belongs
      -- among the standard naturals.

      h-ordx-equals-1H : FiniteGroup.power H# h (X-order x) ≡ 1H
      h-ordx-equals-1H = step-6  where
        step-1 : ι (FiniteGroup.power H# h (X-order x)) (Group.power G# (emb x) (X-order x))
        step-1 = ι-hn-xn (X-order x) (st-fun _ _ X-order x st-X-order st-x)
```

```
          step-2 : PeriodicGroup.power X# x (X-order x) ≡ 1X
          step-2 = PeriodicGroup.order-identity X# x
          step-3 : emb (PeriodicGroup.power X# x (X-order x)) ≡ 1G
          step-3 = tran (cong emb step-2) emb-identity
          step-4 : Group.power G# (emb x) (X-order x) ≡ 1G
          step-4 = tran (sym (emb-power x (X-order x))) step-3
          step-5 : ι (FiniteGroup.power H# h (X-order x)) 1G
          step-5 = transport* step-4 {λ z → ι (FiniteGroup.power H# h (X-order x)) z}
                     step-1
          step-6 : FiniteGroup.power H# h (X-order x) ≡ 1H
          step-6 = Map-unique-Source 1G st-1G (FiniteGroup.power H# h (X-order x))
                     step-5 1H (Map-preserves-unit st-G#)

        h-order : FiniteGroup.order H# h ≤ X-order x
        h-order =
          ℕ-induction {_} {λ n → X-order x ≡ n → FiniteGroup.order H# h ≤ n} case-A case-B (X-
order x) refl where
          case-A : X-order x ≡ 0 → FiniteGroup.order H# h ≤ 0
          case-A ordx-equals-0 = absurd (PeriodicGroup.order-nonzero X# x ordx-equals-0)
          case-B : ∀ (k : ℕ) → (X-order x ≡ k → FiniteGroup.order H# h ≤ k) →
                              X-order x ≡ suc k → FiniteGroup.order H# h ≤ (suc k)
          case-B k ihyp ord-x-equals-suc-k = step-2 where
            step-1 : FiniteGroup.power H# h (suc k) ≡ 1H
            step-1 = transport ord-x-equals-suc-k {λ n → FiniteGroup.power H# h n ≡ 1H} h-ordx-
equals-1H
            step-2 : FiniteGroup.order H# h ≤ suc k
            step-2 = FiniteGroup.order-minimal H# h k step-1

        open import IST.NewmansTheorem

        -- We apply the corollary of Newman's theorem to obtain a standard ν
        -- such that for any finite group G, g∈G and faithful discrete action
        -- @ of G on the manifold M, we can find some n < ord(g) and m'∈M
        -- such that gⁿ@m' is ν-far from m'.
        -- In particular, we shall find n < ord(h) and m'∈M such that
        -- hⁿ@m' is ν-far m'. Since ord(h) is standard, so is n.

        by-newman-1 : ∃ λ (ν : ℝ) → (0r < ν) ∧ (
          ∀ (G : FiniteGroup) →
          ∀ (g : FiniteGroup.Carrier G) →
          ∀ (A : DiscreteAction G M#') →
          (g ≡ FiniteGroup.identity G → ⊥) →
          (∀ (x : FiniteGroup.Carrier G) → (x ≡ FiniteGroup.identity G → ⊥) →
            ∃ λ (m : M) →
            DiscreteAction.Map A x m ≡ m → ⊥) →
          ∃ λ (n : ℕ) → ∃ λ (m : M) →
          (n ≤ FiniteGroup.order G g) ∧
          (ν < distance m (DiscreteAction.Map A (FiniteGroup.power G g n) m)))
        by-newman-1 =
          NewmanSpace.newman-constant M# , (NewmanSpace.isPositive M#) ,
          (λ G g A p → NewmanSpace.isNewmanConstant M# G g p A) -- newman-corollary M#

        ν : ℝ
        ν = proj₁ by-newman-1

        st-ν : st ν
        st-ν = st-fun _ _ NewmanSpace.newman-constant M# st-NewmanSpace-newman-constant st-M#

        positive-ν : 0r < ν
        positive-ν = proj₁ (proj₂ by-newman-1)

        by-newman-2 :
          ∀ (G : FiniteGroup) →
          ∀ (g : FiniteGroup.Carrier G) →
          ∀ (A : DiscreteAction G M#') →
          (g ≡ FiniteGroup.identity G → ⊥) →
          (∀ (x : FiniteGroup.Carrier G) → (x ≡ FiniteGroup.identity G → ⊥) →
            ∃ λ (m : M) →
            DiscreteAction.Map A x m ≡ m → ⊥) →
          ∃ λ (n : ℕ) → ∃ λ (m : M) →
          (n ≤ FiniteGroup.order G g) ∧
          (ν < distance m (DiscreteAction.Map A (FiniteGroup.power G g n) m))
        by-newman-2 = proj₂ (proj₂ by-newman-1)
```

```
by-newman-3 :
  ∃ λ (n : ℕ) → ∃ λ (m : M) →
  (n ≤ FiniteGroup.order H# h) ∧
  (ν < distance m (act (FiniteGroup.power H# h n) m))
by-newman-3 = by-newman-2 H# h A# h-not-id act-faithful

n' : ℕ
n' = proj₁ by-newman-3

m' : M
m' = proj₁ (proj₂ by-newman-3)

n'-less-than-order : n' ≤ FiniteGroup.order H# h
n'-less-than-order = proj₁ (proj₂ (proj₂ by-newman-3))

st-n' : st n'
st-n' = bounded-st (X-order x) (st-fun _ _ X-order x st-X-order st-x) n'
  (≤-tran n' (FiniteGroup.order H# h) (X-order x) n'-less-than-order h-order)

hn' : H
hn' = FiniteGroup.power H# h n'

hn'm'-ν-far-from-m' : ν < distance m' (act hn' m')
hn'm'-ν-far-from-m' = proj₂ (proj₂ (proj₂ by-newman-3))

hn'm'-not-near-m' : nearby (act hn' m') m' → ⊥
hn'm'-not-near-m' hn'm'-near-m' = <-asym-1 _ _ step-3 refl where
  step-1 : ν < distance (act hn' m') m'
  step-1 = transport (symmetry m' (act hn' m')) {λ z → ν < z} hn'm'-ν-far-from-m'
  step-2 : distance (act hn' m') m' < ν
  step-2 = hn'm'-near-m' ν st-ν positive-ν
  step-3 : ν < ν
  step-3 = <-tran ν (distance (act hn' m') m') ν step-1 step-2

-- The manifold element m'∈M might not satisfy standardness. Fortunately, by the
-- compactness of M, we can find a standard neighbor m∈M.

m : M
m = proj₁ (compact m')

st-m : st m
st-m = proj₁ (proj₂ (compact m'))

m-near-m' : nearby m m'
m-near-m' = proj₂ (proj₂ (compact m'))

hn'm-near-hn'm' : nearby (act hn' m) (act hn' m')
hn'm-near-hn'm' = S-uniform-continuity hn' m m' m-near-m'

hn'm-not-near-m : nearby (act hn' m) m → ⊥
hn'm-not-near-m hn'm-near-m = hn'm'-not-near-m' step-3 where
  step-1 : nearby (act hn' m') (act hn' m)
  step-1 = symmetric _ _ hn'm-near-hn'm'
  step-2 : nearby (act hn' m') m
  step-2 = transitive _ _ _ step-1 hn'm-near-m
  step-3 : nearby (act hn' m') m'
  step-3 = transitive _ _ _ step-2 m-near-m'

-- By the standardness of m, we have xⁿ@m near hⁿ@m, and since
-- hⁿ@m lies far from m, so does xⁿ@m. Hence, xⁿ@m ≠ m.

xn' : X
xn' = PeriodicGroup.power X# x n'

st-xn' : st xn'
st-xn' = st-fun _ _ (PeriodicGroup.power X# x) n'
  (st-fun _ _ (PeriodicGroup.power X#) x
  (st-fun-d _ _ PeriodicGroup.power X# st-PeriodicGroup-power st-X#) st-x) st-n'

xn'm-near-hn'm : nearby (xact xn' m) (act hn' m)
xn'm-near-hn'm = act'-lemma (emb xn') st-emb-xn' m st-m hn' ι-hn'-xn' where
  st-emb-xn' : st (emb xn')
  st-emb-xn' = st-fun _ _ emb xn' st-emb
    (st-fun _ _ (PeriodicGroup.power X# x) n'
    (st-fun _ _ (PeriodicGroup.power X#) x
```

```
          (st-fun-d _ _ PeriodicGroup.power X# st-PeriodicGroup-power st-X#) st-x) st-n' )
        ι-hn'-xn' : ι hn' (emb xn')
        ι-hn'-xn' = transport* (sym (emb-power x n')) {λ z → ι hn' z} (ι-hn-xn n' st-n')

      xn'm-not-near-m : nearby (xact xn' m) m → ⊥
      xn'm-not-near-m xn'm-near-m = hn'm-not-near-m step-2 where
        step-1 : nearby (act hn' m) (xact xn' m)
        step-1 = symmetric _ _ xn'm-near-hn'm
        step-2 : nearby (act hn' m) m
        step-2 = transitive _ _ _ step-1 xn'm-near-m

      xn'm-not-equals-m : xact xn' m ≡ m → ⊥
      xn'm-not-equals-m xn'm-equals-m = xn'm-not-near-m xn'm-near-m where
        xn'm-near-m : nearby (xact xn' m) m
        xn'm-near-m = transport* (sym xn'm-equals-m) {λ z → nearby z m} (reflexive m)

      -- From xⁿ@m ≠ m, it follows that x@m ≠ m. We chose x arbitrarily, so we get
      -- faithfulness.

      xm-not-equals-m : xact x m ≡ m → ⊥
      xm-not-equals-m xm-equals-m =
        xn'm-not-equals-m (PeriodicDiscreteAction.power-faithful X-Action x m n' xm-equals-m)

      exists-xm-not-equals-m : ∃* λ (m : M) → (st m) *∧* internal (xact x m ≡ m → ⊥)
      exists-xm-not-equals-m = m , st-m , fromInternal xm-not-equals-m
    open Given

    faithfulness-st : ∀ (x : X) → st x → (x ≡ 1X → ⊥) →
                      ∃* λ (m : M) → (st m) *∧* internal (xact x m ≡ m → ⊥)
    faithfulness-st = exists-xm-not-equals-m

    faithfulness-var : ∀ (x : X) → st x → ∃* λ (m : M) → (st m) *∧* internal ((x ≡ 1X → ⊥) →
xact x m ≡ m → ⊥)
    faithfulness-var x st-x = by-cases* _ case-1 case-2 (excluded-middle (x ≡ 1X)) where
      zm : M
      zm = NewmanSpace.inhabitant M#
      st-zm : st zm
      st-zm = st-fun-d _ _ NewmanSpace.inhabitant M# st-NewmanSpace-inhabitant st-M#
      case-1 : x ≡ 1X →
               ∃* λ (m : M) → (st m) *∧* internal ((x ≡ 1X → ⊥) → xact x m ≡ m → ⊥)
      case-1 x-equals-1 = zm , st-zm , fromInternal (λ x-neq-1 → absurd (x-neq-1 x-equals-1))
      case-2 : (x ≡ 1X → ⊥) →
               ∃* λ (m : M) → (st m) *∧* internal ((x ≡ 1X → ⊥) → xact x m ≡ m → ⊥)
      case-2 x-neq-1 = vm , st-vm , fromInternal (λ z → step-2) where
        step-1 : ∃* (λ m₁ → st m₁ *∧* internal (xact x m₁ ≡ m₁ → ⊥))
        step-1 = faithfulness-st x st-x x-neq-1
        vm : M
        vm = proj₁ step-1
        st-vm : st vm
        st-vm = proj₁ (proj₂ step-1)
        step-2 : xact x vm ≡ vm → ⊥
        step-2 = toInternal _ (proj₂ (proj₂ step-1))

    faithfulness : ∀ (x : X) → ∃ λ (m : M) → (x ≡ 1X → ⊥) → xact x m ≡ m → ⊥
    faithfulness = ax-Transfer-EI Φ faithfulness-var std-Φ where
      Φ : TransferPred
      Φ = ∀' X λ x → ∃' M λ m → int' ((x ≡ 1X → ⊥) → xact x m ≡ m → ⊥)
      std-Φ : st X *∧* (∀ (a : X) → st a → st M *∧*
        (∀ (e : M) → st e → st ((a ≡ 1X → ⊥) → xact a e ≡ e → ⊥)))
      std-Φ =
        st-X , λ a st-a → st-M , λ e st-e → st-→ (a ≡ 1X → ⊥)
        (st-→ (a ≡ 1X) (st-fun _ _ (_≡_ a) 1X (st-fun _ _ _≡_ a st-≡-full st-a) st-1X) ⊥ st-⊥)
(xact a e ≡ e → ⊥)
        (st-→ (xact a e ≡ e) (st-fun _ _ (_≡_ (xact a e)) e (st-fun _ _ _≡_ (xact a e)
        st-≡-full (xact-st-valued a st-a e st-e)) st-e) ⊥ st-⊥)
```