Exact Real Search:
Formalised Optimisation and Regression in Constructive Univalent Mathematics

by

Todd Waugh Ambridge

A thesis submitted to the University of Birmingham for the degree of
Doctor of Philosophy

School of Computer Science
College of Engineering and Physical Sciences
University of Birmingham
14th July 2023

Abstract

The real numbers are important in both mathematics and computation theory. Computationally, real numbers can be represented in several ways; most commonly using inexact floating-point data-types, but also using exact arbitrary-precision data-types which satisfy the expected mathematical properties of the reals. This thesis is concerned with formalising properties of certain types for exact real arithmetic, as well as utilising them computationally for the purposes of search, optimisation and regression.

We develop, in a constructive and univalent type-theoretic foundation of mathematics, a formalised framework for performing search, optimisation and regression on a wide class of types. This framework utilises Martín Escardó's prior work on searchable types, along with a convenient version of ultrametric spaces — which we call closeness spaces — in order to consistently search certain infinite types using the functional programming language and proof assistant Agda.

We formally define and prove the convergence properties of type-theoretic variants of global optimisation and parametric regression, problems related to search from the literature of analysis. As we work in a constructive setting, these convergence theorems yield computational algorithms for correct optimisation and regression on the types of our framework.

Importantly, we can instantiate our framework on data-types from the literature of exact real arithmetic. The types for representing real numbers that we use are the ternary signed-digit encodings and a simplified version of Hans-J. Boehm's functional encodings. Furthermore, we contribute to the extensive work on ternary signed-digits by formally verifying the definition of certain exact real arithmetic operations using the Escardó-Simpson interval object specification of compact intervals.

With this instantiation, we are able to perform our variants of search, optimisation and regression on representations of the real numbers. These three processes comprise our framework of *exact real search*; we close the thesis by providing some computational examples of this framework in practice.

I dedicate this thesis to my father, Michael Andrew Ambridge.
I miss the chats that we could have had about Computer Science.

Acknowledgements

The writing of this thesis could not have been done without significant academic, technical, motivational and emotional input from a large number of people. This short note cannot capture everyone that I am thankful for, and so I would simply like to thank every colleague, friend and family member who has supported me on this academic journey, which I have found both terribly challenging and hugely rewarding.

Most of all, I would first like to thank my supervisors, Professors Dan Ghica and Martín Escardó. To Dan, I thank you for our many enjoyable and enlightening conversations, for always having faith in me and my work, and for always being a motivating and encouraging supervisor. To Martín, I thank you for educating me in constructive mathematics and type theory, for being rigorous in your expectations of me and my work, and for going completely above-and-beyond in your role as a second supervisor. I look forward to continuing to work with both of you, as a fellow teacher and researcher, in the future.

I also thank John Longley and Ulrich Berger for agreeing to examine this thesis and for giving me an interesting, rigorous and engaging viva. Your comments have improved this thesis tenfold, and for that I am exceedingly grateful.

I next thank three peers of mine over the past four years: George Kaye, Tom de Jong and Andrew Sneap. George and I have been friends since our undergraduate days, and embarking on parallel academic journeys has meant I am a constant companion of his wit, charm and pedantry — of which he has all three in great measure. Tom has supported me ever since we started our Ph.D.s at the same time, and I'm very happy that we became great friends — and his patience and thoughtfulness in answering my countless questions has been a tremendous help. Andrew and I have collaborated throughout the latter half of my Ph.D., and I have found our collaboration immensely rewarding — I have enjoyed my transition from his co-supervisor, to his colleague, to his friend. I also want to thank these three for their contributions to this thesis: Tom for his fantastic template (sorry for butchering it with better line spacing), George for much-needed technical support and Andrew for the time we have spent in JAVA together.

Further, I thank the University of Birmingham's School of Computer Science *Theory Group* for the continual help and support, and moreover the sense of community and belonging over the past four years. The ill-named Lab Lunch has proved a supportive setting for workshopping ideas and getting feedback, but the help has been more keenly felt by the fact I can call many of you friends. In particular, I would like to thank Ayberk Tosun, Paul Levy, Sonia Marin, Eric Finster, Anupam Das, Paul Taylor and Achim Jung.

I would further like to thank my office-mates Qamar Natsheh and Mubashir Ali, for their kind words and support especially in these last few weeks. In the wider academic community, I would like to thank the CCC research community, especially Alex Simpson with whom I had some very interesting discussions. I also give my thanks to the DOÜVK community at the School of Computer Science for their camaraderie — thank you Jacqui, Matthew, Jon, Anna, Bruno, Tobias, Charlotte, Charlie and Yan.

Outside of my studies, I thank all of my friends who have supported me: especially my brotherhood of school friends, Callam, James, Ollie, Owen, Will and Todd, and the friends I have made since moving to Birmingham, Freddie, Fran, Charlie, Cathy, Adam and Tom. Thank you also to my family: to Jody, to Colin, to Nana Vee, and to my lovely girls Cocoa, Lola and Sookie.

There are two people left to thank, who have been there in my best and worst moments, and who have helped more than I can express in words. To Alice, I say thank you for your encouragement and support, for your love and companionship, for putting up with me and for being my best friend — I look forward to the next chapter of our lives together, where hopefully the word "thesis" does not feature so much. Finally, to my mum, I simply say thank you for everything you have ever given me, which is everything I have.

Table of Contents

Chapter 1

Introduction

The real numbers are a fundamental structure in a variety of fields such as real analysis, calculus, optimisation theory and regression analysis [Lib08; YS09]. The reals are also important computationally; the re-burgeoning field of machine learning, for example, is heavily dependent on floating-point representations of real numbers, wherein they are used to model continuous parameters of artificial neural networks [WCB+18]. Arithmetic on floating-point numbers is incredibly, and increasingly, efficient [WK19], but not without fault: floating-point methods represent real numbers as one of finitely-many dyadic rationals[1], leading to representation and calculation errors [JR21; Gol91]. There are, however, alternative approaches to real number computation, such as interval arithmetic — wherein functions are evaluated by their behaviour on intervals of the real numbers [AM00] — or arbitrary-precision '*exact real*' computation [BCRO86].

Using exact real computation, any (computable) real number can be represented to any degree of precision, and manipulated correctly with respect to the represented real number [Tur37; BCRO86]. This means that, in situations where "floating-point algorithms can lead to completely erroneous results ... exact real number computation provides guaranteed correctness" [Sim98]. The applications of exact real computation have thus far largely focused on arithmetic; indeed, perhaps the most omnipresent use of exact reals is in the built-in Android smartphone calculator app [Boe17]. In this thesis, we investigate another application of exact real computation: the construction of formally verified algorithms for search, optimisation and regression.

Search has previously been performed on exact reals. For example, Simpson uses

[1]A dyadic rational is a rational number of form $\frac{n}{2^i}$.

an earlier algorithm by Berger to search for the global minimum *value* of a real-valued function in the compact interval $[-1, 1]$ [Sim98; Ber90]. The representation of $[-1, 1]$ utilised by Simpson is the *ternary signed-digit encodings*, which has been extensively explored in the literature of exact real arithmetic [Plu98; Di 93; Ber09]. More recently, Escardó implemented search algorithms using the ternary signed-digits in Haskell [Esc11b], as part of his wider work on *searchable sets* [Esc08].

In this thesis, we use searchable types in order to construct algorithms for global optimisation and parametric regression on the ternary-signed digits. Global optimisation is the problem of finding a global minimum *argument* to a (usually real-valued) function in a compact interval, while parametric regression is the problem of finding parameters that fit a (usually real-valued) model function to some reference data [Lib08; YS09].

In order to build formally verified algorithms, we will work in a *constructive* foundation for performing mathematics and computation in tandem. When working constructively, a mathematician cannot rely on the law of excluded middle, proofs by contradiction or principles of omniscience[2] [BB12]. Instead, in order to show the truth of a claim, a constructivist must show exactly how it holds. In the case where the claim relates to an algorithm, they must genuinely construct that algorithm. A constructive approach to mathematics, therefore, provides additional challenges; but there are also advantages: one could extract the algorithm from the constructive proof, and use it computationally with the knowledge it satisfies the claim. For this purpose, we develop our work entirely within the functional programming language, and proof assistant, Agda. By working in Agda — which itself is based on Martin-Löf 's constructive type theory (MLTT) [BDN09; NPS90] — we effectively *program* formalised type-theoretic mathematical definitions, statements and proofs, and can immediately extract and run algorithms based on these structures' computational content. Our constructive, type-theoretic formalisation is defined within Escardó 's Agda library for univalent mathematics TypeTopology [Esc23].

We first develop a framework for search, optimisation and regression on a wide class of types, using the work on searchable types already defined by Escardóin TypeTopology [Esc11a]. Then, we formalise the ternary-signed digit encodings and verify many of their exact real arithmetic operations in Agda; similar to di Gianantonio's formal verification of the ternary signed-digits in Coq [CD06]. For this purpose, we first formalise a specification of $[-1, 1]$, namely the Escardó -Simpson interval object [ES01]. We instantiate our general framework on the ternary signed-digits, allowing us to extract

[2]Such as LPO, WLPO or LLPO, which are used at various points in this thesis to show that something is contructively invalid.

algorithms for search, optimisation and regression on this representation. Following this primary instantiation, in order to investigate the framework's applicability (and whether or not in can lead us towards efficient practical implementations) we utilise another type for representing exact real numbers: a simplified definition of the Boehm encodings, which are used today — nearly thirty-years on from their original definition — in the ANDROID calculator app [Boe99; Boe20].

1.1 Thesis outline and key contributions

In Chapter 2, we introduce the constructive and univalent type-theoretic foundation of mathematics in which we build our formal framework for search, optimisation and regression. We will establish the notation of the thesis informally, based on the syntax of AGDA.

In Chapter 3, we review and define the two key mathematical concepts of the thesis: *searchability* and *uniform continuity*. We review the literature on *searchable types*, and their current status in constructive type theory as implemented by Martín Escardó, and contribute an extension of his work in order to be able to consistently search certain infinite types in AGDA by introducing uniform continuity on a convenient version of ultrametric spaces that we call *closeness spaces*. We give a version of the totally bounded property for closeness spaces, and show that a variety of types yield closeness spaces. The key technical contribution of this section is the formalised result which shows these *uniformly continuously searchable types* are closed under countable products (Theorem 3.3.14).

In Chapter 4, we use uniformly continuously searchable closeness spaces to define our formal convergence properties of global optimisation and parametric regression on a wide class of types. For this purpose, we introduce *approximate linear preorders*, which approximately order elements of closeness spaces. The key contribution of this section — the statement of the type-theoretic variants of global optimisation (Theorem 4.1.26) and parametric regression (Theorems 4.2.6, 4.2.9 and 4.2.10) — is methodological rather than technical, as the proofs of their convergence follow naturally from the concepts we have introduced.

In Chapter 5, we review and define within AGDA two types for representing real numbers: ternary signed-digit encodings and ternary Boehm encodings. On the former, we formally verify exact real arithmetic operations (namely, negation, binary and infinitary midpoint and multiplication) using the Escardó -Simpson interval object specification of closed intervals — which we also review and formalise in this section. On the latter, we define the type in AGDA, prove the correctness of its structure and show how it yields representations of compact intervals that we can then use for search. The key technical contributions of this section are:

- The Agda formalisation of the Escardó-Simpson interval object specification of closed intervals (Section 5.1),
- The Agda formalisation of the ternary signed-digit encodings and their aforementioned arithmetic operations (Section 5.2),
- The formal verification in Agda that these operations for exact real arithmetic on the ternary-signed digits are correct with respect to the specification of those operations on the interval object (Theorems 5.2.10, 5.2.21, 5.2.32 and 5.2.36).

In Chapter 6, we bring our formal framework full-circle by instantiating it on these two types for representing real numbers. Example evaluations of algorithms for search, optimisation and regression — either extracted from Agda or implemented in Java — are then given to show the use of the framework in practice. A contribution of this section is the formal result that the arithmetic operations we define on the ternary signed-digit encodings are uniformly continuous (Section 6.1.2).

Finally, in Chapter 7, by way of conclusion we discuss some further avenues for this line of work.

1.1.1 Reading the formal proofs of this thesis

A chief contribution of this thesis is that most of its mathematical content — both from the literature and our own contributions — is formalised in Agda within the library TypeTopology [Esc23]. We describe the constructive and univalent philosophy of TypeTopology in the intoduction to Chapter 2.

The reader is invited to explore our Agda formalisation by 'clicking' on the symbols at the top left of each mathematical environment. This will take them directly to the Agda function which formalises that definition or proof. The different files of the library are described in Appendix A.

For the purpose of clarity, we use three different symbols:

- The *library book* symbol 🕮 denotes a statement we are recalling from the literature and which we have *not* formalised, nor does a formalisation appear within TypeTopology,
- The *topological donut* ◎ symbol denotes a statement formalised within TypeTopology, usually by Martín Escardó, but sometimes by another collaborator,
- The *rune of Gandalf* ᚡ symbol denotes a statement formalised for this thesis by the author. Sometimes, this will be a repetition from the literature or TypeTopology, but in other cases this will be one of our main contributions, as outlined in Section 1.1.

Chapter 2

Constructive Univalent Type Theory via Agda

We wish to perform mathematics in a way that supports computer programming by default, in order to extract computational content from our mathematical proofs. As discussed in Chapter 1, this means that we will be utilising a *constructive* approach to mathematics. But we want to go further; we do not wish to just support programming but to actively program formal mathematics. For this purpose, we work formally in a constructive and univalent foundation of mathematics: the variation of Martin-Löf constructive type theory (MLTT) provided by the functional programming language and proof assistant Agda [Mar75; BDN09]. More specifically, we work in TypeTopology, an Agda library by Martín Escardó and a growing number of collaborators interested in formalising both new and previous theorems in univalent mathematics [Esc23]. This thesis is second only to Tom de Jong's recent thesis on *Domain Theory in Constructive and Predicative Univalent Foundations* in having the majority of its results formalised within TypeTopology [dJon23].

The philosophy of this thesis is aligned with that of TypeTopology, the full extent of which is given on the library's webpage [Esc23]. In particular, we work in a small version of MLTT — which we introduce in Section 2.2 — and we adopt the univalent approach to mathematics introduced by Voevodsky and popularised by *The HoTT Book* [Uni13]. This latter point means that, even when not invoking the univalence axiom itself, we utilise the terminology and the perspective of univalent mathematics, which we detail in Section 2.3. Our framework is hence compatible with other formalisations of univalent mathematics, such as the UniMath library for Coq [VAG+] — unlike UniMath however, we do not assume the propositional resizing axiom. When we *do* use axioms such as function extensionality or univalence, we make them explicit parameters to those proofs or modules which use them. Finally, we restrict ourselves to those features of Agda which allow us to remain consistent, and avoid inconsistent assumptions such as the 'type-in-type' axiom (which assumes the type of all types is an element of itself).

We begin this chapter with a brief introduction to MLTT, before recalling concepts of this theory and explaining how they are written in general Agda. We then introduce the aspects of univalent mathematics that we utilise, before recalling some fundamental concepts that are used throughout the following chapters of the thesis.

2.1 A brief introduction to type theory

A *type theory* is a set of rules for formally reasoning about the behaviour of a *system of terms* such as a logical calculus, foundation of mathematics, philosophical theory, or programming language [Pie02]. Informally, the rules assign to each term a *type* that is used to determine the *behaviour* of such terms, such as which methods of the theory can manipulate them. These methods, such as *functions*, are themselves terms of more complicated types formed from simpler types using *type families*.

Early type theories were developed by Russell in the 1900s as foundations of mathematics alternative to those built within naive set theory, which he famously showed yielded inherent paradoxes [WR27]. Soon thereafter, Church's *simply typed lambda-calculus* utilised Ramsey's *simple theory of types* to ensure that each term of the calculus is well-typed [Chu40]. Later, in 1972, Per Martin-Löf introduced MLTT as a "full scale system for formalising intuitionistic mathematics" [Mar75]. Martin-Löf's intention to fully-formalise Bishop-style constructive mathematics (discussed in Chapter 1) had clear motivations: the internal computation rules of type theories are happily married to the ability to extract computational content from constructive proofs.

In MLTT, a mathematical proposition is interpreted as a type. An *element* (i.e. a *program*) of such a type is considered a proof of the proposition, and can therefore be computed to yield a constructive witness of that proof. These types are built from type families which interpret the connectives of intuitionistic logic, and are often *dependent* on types that model more fundamental mathematical structures (such as the natural numbers) or, indeed, other propositions. This is known as the *propositions as types* interpretation [Uni13]. This interpretation means that two proofs $a_1, a_2 : A$ of the same mathematical statement may be very different programs, and this difference is often relevant to proving later corollaries of that statement — this is called *proof relevance*.

The most significant dependent type family of MLTT is the *identity type family*, which is used to interpret statements about term equality. We can say that two elements $a_1, a_2 : A$ are equal if we can construct an element of type $(a_1 = a_2)$ [Mar75]. This makes our foundation of mathematics very rich: we can begin to reason directly about the equality of mathematical objects/programs in the way that a mathematician/programmer would naturally do so. Matching equality in MLTT more and more closely to a natural notion of mathematical equality is the primary concern of the field of *univalent type theory* [Uni13]; an area that this thesis operates within, and which we shall return to later in Section 2.3.

2.2 AGDA notation for constructive type theory

There are already a variety of formal introductions to MLTT's types and type families, from various perspectives and levels of introduction (for example, see [MS84; Gam09] or the Appendices of [Uni13]). We assume the reader has some familiarity with type theory, and in this section instead recall the constructions and concepts of MLTT and show how to write them in AGDA. In the remainder of the thesis, however, we will write mathematics informally, in the style of *The HoTT Book* [Uni13]. This section can therefore be thought of as the documentation required to be able to read this thesis' AGDA formalisation.

2.2.1 Type universes

Recall that a type universe is a type whose elements are themselves types [Mar75]. MLTT utilises a countable sequence of *type universes*, $\mathcal{U}_0 : \mathcal{U}_1 : \mathcal{U}_2, \ldots$; the integer indices of these type universes are sometimes called *universe levels* [BCDE23].

In AGDA, type universes are explicitly stratified into countably-many universe levels. There is therefore a lowest level `lzero` and a successor operation `lsuc`. This successor function forms a semilattice, whose join is given as the binary operation `_⊔_`.

```
lzero : Level
lsuc  : (l : Level) → Level
_⊔_   : (l₁ l₂ : Level) → Level
```

As `Level` is a semilattice, the supremum function is associative $(u \sqcup (v \sqcup w) \equiv (u \sqcup v) \sqcup w)$, commutative $(u \sqcup v \equiv v \sqcup u)$ and idempotent $(u \sqcup u \equiv u)$ — furthermore, the successor operation is a homomorphism (`lsuc` $(u \sqcup v) \equiv$ `lsuc` $u \sqcup$ `lsuc` v). Note that the preceding equations are *definitional equalities* and not equalities induced by the identity type former (introduced in Section 2.2.7).

Each universe level `i` is mapped to a type universe $\texttt{Set}_\texttt{i}$ such that $\texttt{Set}_\texttt{i} : \texttt{Set}_{\texttt{lsuc i}}$, meaning there is indeed, as in MLTT, a countable sequence of type universes. Furthermore, there is a type universe `Setω` that is larger than every type universe, but is not part of the hierarchy — for example, the expression $(i : \texttt{Level}) \to \texttt{Set}_i$ has type `Setω` [Imp15].

The notation of type universes in TYPETOPOLOGY aligns the above closer to that of MLTT [Esc13b]:

```
open import Agda.Primitive public
  using (_⊔_)
  renaming (lzero to 𝓤₀
           ; lsuc to _⁺
           ; Level to Universe
           ; Setω to 𝓤ω
           )
```

`Set` is renamed to `Type`, and universes are mapped to type universes using the 'dot function'. We prefer this notation as not all types are sets in MLTT, and furthermore the very small difference in syntax between universes $\mathcal{U}$ and type universes $\mathcal{U}^{\cdot}$ allows the reader of a TYPETOPOLOGY file to align these concepts in their mind, as we do when working in MLTT [Esc19].

Formally, we follow these conventions of TYPETOPOLOGY; informally, universes are implicit in most of our work. Where necessary, we use $\mathcal{U}, \mathcal{V}, \ldots$ to range over type universes.

2.2.2 Function and Π-types

Functions are built-in to AGDA, and interpret statements that are universally quantified over a type. We give the curried projection functions for every type (in every universe) as example function definitions:

```
fst      : {X : 𝓤 ̇} {Y : 𝓥 ̇} → X → Y → X
fst {𝓤} {𝓥} {X} {Y} x y = x
```

```
snd     : {X : 𝓤 ̇} {Y : 𝓥 ̇} → X → Y → Y
snd {𝓤} {𝓥} {X} {Y} x y = y
```

Note that implicit arguments (i.e. the types and the universes they are elements of) are given in braces in the type definition, and can be accessed using braces in the function definition.

Dependent functions (elements of Π-types), where the type of the output depends on that of the input, are also built-in using the same syntax. In TYPETOPOLOGY, we match this syntax to that of MLTT by defining a universe-valued dependent function Π as below (although, for brevity of code, we almost always utilise the usual AGDA notation of Π-types):

```
Π : {X : 𝓤 ̇} (Y : X → 𝓥 ̇) → 𝓤 ⊔ 𝓥 ̇
Π {𝓤} {𝓥} {X} Y = (x : X) → Y x
```

Elements of Π-types $f : \Pi_{(x:X)}\,(Y\,x)$ are therefore dependent functions that, on input of $x : X$, output $f(x) : Y\,x$ — usual functions are just dependent functions where the type family Y is constant.

2.2.3 Natural number and integer types

The *natural numbers* are defined inductively using the `data` keyword:

```
data ℕ : 𝓤₀ ̇ where
 zero : ℕ
 succ : ℕ → ℕ
```

A type defined by `data` is given by a list of its *constructors*; here, ℕ is defined using the Peano inductive definition. Note, therefore, that the constructor succ : ℕ → ℕ is itself an AGDA function from natural numbers to natural numbers. We will write naturals in the expected shorthand way, e.g. $0 :=$ zero and $n + 1 :=$ succ n.

In AGDA, functions on inductive types can be defined by *pattern matching*, i.e. by considering the function's output for each of its constructors. As an example, we define addition on the natural numbers as an infix operation by pattern matching:

```
_+ℕ_ : ℕ → ℕ → ℕ
0      +ℕ y = y
succ n +ℕ y = succ (n +ℕ y)
```

The *integers* $\mathbb{Z}$ are also defined using `data`. The two constructors are for negative and non-negative numbers:

```
data ℤ : 𝓤₀ ̇ where
 pos     : ℕ → ℤ
 negsucc : ℕ → ℤ
```

Again, we will write integers in the expected shorthand way; e.g. $0 :=$ pos 0, and $-6 :=$ negsucc 5.

2.2.4 Unit, empty and negated types

Recall, by the propositions-as-types interpretation, that any *pointed* type (i.e. a type that we can exhibit an element of) represents the trivially true proposition. The *unit type* $\mathbb{1}$, which has one constructor and which we often use for representing truth, is defined as follows:

```
data 𝟙 {𝓤} : 𝓤 ̇ where
  ⋆ : 𝟙
```

The *empty type* $\mathbb{0}$, which has no constructors, can also be defined using `data`:

```
data 𝟘 {𝓤} : 𝓤 ̇ where
```

Note that there are unit and empty types in every universe.

Recall that negation is interpreted in MLTT by empty-valued functions. We therefore define the type family $\neg X$ given any type X as follows:

```
¬ : (X : 𝓤 ̇) → 𝓤 ̇
¬ {𝓤} X = X → 𝟘
```

Exhibiting an element of $\neg X$ therefore amounts to defining a function that proves any element of X leads to false.

In the opposite direction, by the *principle of explosion* any proof of false itself implies anything. This can be defined in AGDA as a dependent function by pattern matching:

```
𝟘-elim : {X : 𝓤 ̇} → 𝟘 → X
𝟘-elim {𝓤} {X} ()
```

The special pattern `()` is used here by AGDA to denote that there is no pattern for that element (i.e. the element of type 𝟘), and thus we do not have to define the function for that case.

2.2.5 Disjoint union of types

Given two types, we can form their *disjoint union type*, which has one constructor for each side of the disjunction:

```
data _+_ {𝓤} {𝓥} (X : 𝓤 ̇) (Y : 𝓥 ̇) : 𝓤 ⊔ 𝓥 ̇ where
 inl : X → X + Y
 inr : Y → X + Y
```

These types represent disjoint statements in intuitionistic logic, as we must provide a witness as to which side of the disjunction holds. Furthermore, the form of disjunction that disjoint union types represent is *inclusive*, and yields a direct answer as to which of the two statements holds — even if both could feasibly hold, the element can only reduce to one side of the disjunction.

Disjoint unions are therefore in general *structure* and not *property*. Indeed, we often use them in the definition of mathematical structures; as an example, the three-point type can be given by a disjunction of the unit type:

```
𝟛 : 𝓤₀ ̇
𝟛 = 𝟙 + 𝟙 + 𝟙
```

We will return to discussion of properties versus structure in Section 2.3.2.

2.2.6 Binary product and Σ-types

Given two types, we can form their *binary product type*, whose elements are pairs. We define these as non-dependent versions of *dependent pairs* (elements of Σ-types), where the type of the second projection of the pair depends on that of the first. Σ-types are defined coinductively using the `record` keyword:

```
record Σ {𝓤} {𝓥} {X : 𝓤 ̇} (Y : X → 𝓥 ̇) : 𝓤 ⊔ 𝓥 ̇ where
  constructor _,_
  field
   pr₁ : X
   pr₂ : Y pr₁
```

Then, binary product types can be easily be defined as following type family:

```
_×_ : 𝓤 ̇ → 𝓥 ̇ → 𝓤 ⊔ 𝓥 ̇
X × Y = Σ {𝓤} {𝓥} {X} (λx.Y)
```

A type defined by `record` is given by a list of its *fields*; in this way, records are Agda's built-in version of Σ-types.

Recall that, as mathematical statements, Σ-types interpret constructive existence: proving there is an element $a\colon A$ which satisfies the statement $B(a)\colon \mathcal{V}$, for a type $A\colon \mathcal{U}$ and type family $B\colon A \to \mathcal{V}$, is performed by constructing an element of type $\sum_{(a\colon A)} (B(a))$. In general, this element can be defined in multiple ways, yielding (by proof relevance) multiple proofs of existence.

Σ-types are therefore in general *structure* and not *property*. They are used to define both collections and the more restricted notion of subtypes. As an example, we give below the collection of even natural numbers $\mathbb{N}_e$:

```
is-even : ℕ → 𝓤₀ ̇
is-even 0 = 𝟙
is-even 1 = 𝟘
is-even (succ (succ n)) = is-even n

ℕₑ : 𝓤₀ ̇
ℕₑ = Σ n : ℕ , is-even n
```

An element of $\mathbb{N}_e$ is a dependent pair: a number $n : \mathbb{N}$ and a proof of evenness of type is-even n. Later, in Section 2.3.2 we will see that $\mathbb{N}_e$ is in fact a subtype of $\mathbb{N}$: informally, this is because for any given integer n there is only one proof of evenness (i.e. there is only one element of the type is-even n).

2.2.7 Identity types

Every type has an associated family of *identity types*. Recall that there is only a single constructor for identity types in MLTT — so too in AGDA:

```
data Id {𝓤} (X : 𝓤 ̇ ) : X → X → 𝓤 ̇
  refl : (x : X) → Id X x x
```

The function `refl` allows us only to construct elements of the type `Id X x x`, which we write in TYPETOPOLOGY as `x = x` for any `x : X`[1], leaving the type of `x` implicit.

By default in AGDA, it is the case therefore that Streicher's K axiom is *assumed*: refl x is the only element of $x = x$ [Str93]. However, as this cannot be *proved* in MLTT (or AGDA), we disable its use in TYPETOPOLOGY. But, it *is* the case that Martin-Löf's induction rule for identity types only requires the consideration of elements constructed by refl [MS84]. Martin-Löf called this rule J [Esc19]:

```
J : (X : 𝓤 ̇ ) (A : (x y : X) → x = y → 𝓥 ̇ )
  → ((x : X) → A x x (refl x))
  → (x y : X) (p : x = y) → A x y p
J {𝓤} {𝓥} X A f x x (refl x) = f x
```

From the above, we see that J is defined in AGDA by pattern matching on the element of the identity type $p : x = y$. Once the identity type is pattern matched, x and y are aligned definitionally in AGDA and can be used interchangeably.

Elements of other identity types must be derived from the expected properties of identities, which are defined in AGDA as functions either by using J or by directly pattern matching. We prove the four key identity rules — symmetry, transitivity, function application, and transport — by pattern matching.

[1]Note that we actually use a Unicode 'long equals' symbol in TYPETOPOLOGY, as = is reserved in AGDA for definitions.

```
sym : {X : 𝓤 ̇} {Y : 𝓥 ̇} (x y : X) → x = y → y = x
sym {𝓤} {𝓥} {X} {Y} x x (refl x) = refl x

trans : {X : 𝓤 ̇} {Y : 𝓥 ̇} (x y z : X) → x = y → y = z → x = z
trans {𝓤} {𝓥} {X} {Y} x x x (refl x) = refl x

ap : {X : 𝓤 ̇} {Y : 𝓥 ̇} (f : X → Y) (x y : X) → x = y → f x = f y
ap {𝓤} {𝓥} {X} {Y} f x x (refl x) = refl (f x)

transport : {X : 𝓤 ̇} (A : X → 𝓥 ̇) (x y : X) → x = y → A x → A y
transport {𝓤} {𝓥} {X} A x x (refl x) = id
```

We can consider all of the above functions as *proofs* about identities. Note that when working informally in this thesis we do not detail when we use these rules.

2.3 Univalent mathematics in TypeTopology

Now that we have recalled the key constructions of MLTT using Agda, we change to a more informal approach to mathematics for the remainder of the thesis. This approach will be a mixture of English statements and proofs, peppered with mathematical notation that resembles Agda code. If the reader wishes to read the formalisation of any particular definition or proof, they can be easily accessed by the method described in Section 1.1.1 (i.e. by simply clicking the symbol at the top of the environment).

In this section, we recall principles of the burgeoning field of univalent foundations of mathematics — which is sometimes called *univalent type theory* or *homotopy type theory* — from the point of view of their definition in TypeTopology. In order to get the reader used to our informal notation, we spell out many of the recalled definitions and proofs of this section in English and Agda-like syntax.

Four principles that we will introduce in this section that are worth mentioning in advance are *function extensionality* (labelled [f]), *propositional extensionality* (labelled [p]), *univalence* (labelled [u]) and *propositional truncation* (labelled [t]). These are all axioms in MLTT, and thus we want to be especially clear about where they are used in this thesis. If a proof uses one of these four axioms, we will mark it with its corresponding label — for example, a proof labelled [fp] invokes both function extensionality and propositional extensionality.

2.3.1 Function extensionality

Two (possibly dependent) functions are equivalent if they are behaviourally indistinguishable; i.e. if they are *pointwise-equal*,

Definition 2.3.1. [◎] Two functions $f, g \colon \prod_{(x \colon X)} (Y(x))$ are *pointwise-equal* $f \sim g$ if $f(x) = g(x)$ holds for every $x \colon X$:

$$f \sim g := \prod_{(x \colon X)} f(x) = g(x).$$

Of course, equal functions are pointwise equal.

Lemma 2.3.2. *Every function is pointwise-equal to itself.*

Proof. The proof term is constructed immediately the identity type's constructor (i.e. reflexivity):

$$\begin{aligned}&\sim\text{-refl} : \Pi_{(f \colon \prod_{(x \colon X)} Y(x))} (f \sim f),\\ &\sim\text{-refl}(f, x) := \mathsf{refl}\ (f(x)).\end{aligned}$$

Lemma 2.3.3. [◎] *Given functions $f, g \colon \prod_{(x \colon X)} Y(x)$ such that $f = g$, then $f \sim g$.*

Proof. We utilise the given equality $e \colon f = g$ to build the function

$$\mathsf{transport}(\lambda(h \colon \Pi_{(x \colon X)} Y(x)).f \sim h, e) \colon f \sim f \to f \sim g.$$

Thus, by applying Lemma 2.3.2, the result follows:

$$\begin{aligned}&\mathsf{happly} : \Pi_{(f,g \colon \prod_{(x \colon X)} Y(x))} ((f = g) \to (f \sim g)),\\ &\mathsf{happly}(f, g, e, x) := \mathsf{transport}(\lambda(h \colon \Pi_{(x \colon X)} Y(x)).f \sim h, e, \sim\text{-refl}(X)).\end{aligned}$$

There is no way to prove (for non-trivial types) that pointwise-equal functions are equal in MLTT. In order to be able to treat equivalent functions as equal functions, we can add an extra axiom called *(naive) function extensionality* (later, in Definition 2.3.27 we will introduce the non-naive version).

Definition 2.3.4. [◎] We say that *naive function extensionality* holds for given universes $\mathcal{U}, \mathcal{V}$ when, given $X \colon \mathcal{U}$, $Y \colon X \to \mathcal{V}$ and functions $f, g \colon \prod_{(x \colon X)} Y(x)$, pointwise-equality implies equality:

$$\mathsf{naive\text{-}funext}(\mathcal{U},\mathcal{V}) :\equiv \prod_{(X:\,\mathcal{U})}\ \prod_{(Y:\,X\to\mathcal{V})}\ \prod_{\left(f,g:\ \prod_{(x:\,X)} Y(x)\right)} (f \sim g \to f = g)$$

Although we accept it as a mathematical statement and would like to use it when reasoning about certain functions, function extensionality is independent of MLTT; meaning it is consistent with the type theory but cannot be proved by it. As it is consistent, one can add it as an axiom to the theory and use it in their proofs — in Agda, this amounts to postulating the existence of a proof term $(\mathsf{fe} : \mathsf{naive\text{-}funext})$. If one does this, however, one must be careful about where the postulated term is used. Postulated terms have no computational interpretation; while they may help us to write a proof, we will be unable to extract any data from said proof. In this thesis, function extensionality is sometimes assumed in order to show that our results are correct with regards to certain specifications — but we never assume it when computing those results themselves.

2.3.2 Propositions and unique proofs

Under the propositions as types interpretation described in Section 2.1, the types of MLTT can be viewed as mathematical statements, with elements of those types viewed as proofs of the statement. Hence, by exhibiting different elements of the same type, the truth of a statement can be given in multiple ways. Univalent type theory distinguishes statements that can be true in multiple ways and those that can only be true in one way, calling the latter *(h)-propositions*. A type interprets a proposition if it can only be true in one way; i.e. if every element of the type is identical. These types are sometimes referred to as *subsingletons*, though we simply call them *propositions*. If we agree that propositions are the only types that truly interpret a mathematical proposition, then we are aligning ourselves with the *propositions as some types* interpretation of dependent type theory [Shu12].

Definition 2.3.5. [◎] A type X is a *proposition* if all of its elements are equal:

$$\mathsf{is\text{-}prop}(X) :\equiv \prod_{(x,y:\,X)} (x = y)\,.$$

Lemma 2.3.6. [◎] $\mathbb{1}$ *is a proposition.*

Proof. $\mathbb{1}$ only has one element, $\star\colon \mathbb{1}$; thus, it is trivial to prove that any two $x, y\colon \mathbb{1}$ are equal, as $x = \star = y$.

Lemma 2.3.7. [◎] *$\mathbb{0}$ is a proposition.*

Proof. $\mathbb{0}$ has no elements; thus, given any two $a, b\colon \mathbb{0}$ we can immediately complete the vacuous proof using $\mathbb{0}$-elim.

But it is not the case that these are the only propositions. For every proposition X, it is independent that $(X \simeq \mathbb{1}) + (X \simeq \mathbb{0})$ — a proof of this would be logically equivalent to the classical law of excluded middle for propositions.

+-types that interpret exclusive-or statements preserve the proposition property, meaning the type is now a *property* rather than *structure*.

Lemma 2.3.8. [◎] *If $X\colon \mathcal{U}$ and $Y\colon \mathcal{V}$ are propositions, and $\neg(X \times Y)$, then $X + Y\colon \mathcal{U} \sqcup \mathcal{V}$ is a proposition.*

Proof.

$$\begin{aligned}
&\text{+-is-prop} : \text{is-prop } X \longrightarrow \text{is-prop } Y \longrightarrow \neg(X \times Y) \longrightarrow \text{is-prop } (X + Y)\\
&\text{+-is-prop}(p, q, f, \text{inl } x_1, \text{inl } x_2) &&:= \text{ap}(\text{inl}, p(x_1, x_2))\\
&\text{+-is-prop}(p, q, f, \text{inl } x, \text{inr } y) &&:= \mathbb{0}\text{-elim } (f(x, y))\\
&\text{+-is-prop}(p, q, f, \text{inr } y, \text{inl } x) &&:= \mathbb{0}\text{-elim } (f(x, y))\\
&\text{+-is-prop}(p, q, f, \text{inr } y_1, \text{inr } y_2) &&:= \text{ap}(\text{inr}, q(y_1, y_2))
\end{aligned}$$

Π-types, ¬-types and Σ-types also preserve the proposition property.

Lemma 2.3.9. [◎] *Given $X\colon \mathcal{U}$ and $Y\colon X \longrightarrow \mathcal{V}$ where every $Y(x)\colon \mathcal{V}$ is a proposition, then $\Pi Y\colon \mathcal{U} \sqcup \mathcal{V}$ is a proposition.*

Proof. [f] We need to show that given $f, g\colon \prod Y$ we have $f = g$. We have $f \sim g$ because given any $x\colon X$, $Y(x)$ is a proposition (so therefore $f(x) = g(x)$). The result immediately follows by function extensionality.

Corollary 2.3.10. [◎] *The negation of every type is a proposition.*

Proof. [f] By Lemmas 2.3.7 and 2.3.9.

Lemma 2.3.11. [◎] *Given a proposition $X : \mathcal{U}$ and $Y : X \to \mathcal{V}$ where every $Y(x) : \mathcal{V}$ is a proposition, then $\Sigma Y : \mathcal{U} \sqcup \mathcal{V}$ is a proposition.*

Proof (Sketch). The idea is that, given $(a, p) : \Sigma_{(a : X)} Y(a)$ and $(b, q) : \Sigma_{(b : X)} Y(b)$, we want to show that conclude $(a, p) = (b, q)$ by first showing $a = b$ and $p = q$ — but the first and last equations here do not type check, as the identity type requires both arguments to be of the same type. As X is a proposition, however, we have a proof that $a = b$, and therefore $Y(a)$ and $Y(b)$ *are* indeed the same type (we will arbitrarily choose to call this type $Y(a)$). The result then follows by the fact that $Y(a)$ is a proposition and therefore $p = q$.

If a type P is a proposition, any proof $p : P$ is *unique*. For example, the proof that an integer is even (defined in Section 2.2.6) is unique, because each of its cases is a proposition. Hence, $\mathbb{N}_e$ (which we also defined in Section 2.2.6) is a *subtype* of $\mathbb{Z}$; i.e. it is a unique collection of elements of $\mathbb{N}$. Related to this, we introduce the idea of *embeddings*, which we recall are functions $f : X \to Y$ such that each $y : Y$ is reached by at most one $x : X$.

Definition 2.3.12. [◎] Given types X and Y, a function $f : X \to Y$ is an *embedding* if each fiber of f is a proposition. We write this as follows:

$$\mathsf{is\text{-}embedding}(f) := \prod_{(y : Y)} \mathsf{is\text{-}prop}\left(\sum_{(x : X)} (f x = y)\right).$$

As an example, the function $\mathsf{pr}_1 : \sum_{(x : X)} Y(x) \to X$ is an embedding if Y is a proposition; also, $\mathsf{inl} : X \to X + Y$ and $\mathsf{inr} : Y \to X + Y$ are embeddings.

The subtype that encapsulates all interpretations of truth values in a given universe is that universe's *type of truth values*.

Definition 2.3.13. [◎] The *type of truth values* for a universe $\mathcal{U}$ is the type,

$$\Omega_{\mathcal{U}} := \sum_{(P : \mathcal{U})} (\mathsf{is\text{-}prop}\ P)\ .$$

Note that we usually leave the particular universe implicit, and simply write Ω. In this thesis, we use truth-valued functions often, without explicitly proving (via the above lemmas) that the values of the function are propositions. In the formalisation, this requirement adds additional, but usually straightforward, complexity to the proofs.

2.3.3 Propositional extensionality

Logical equivalence between types is defined in the usual way.

Definition 2.3.14. [◎] Two types X and Y are *logically equivalent* $X \leftrightarrow Y$ if $X \to Y$ and $Y \to X$.

However, given our discussions in the previous subsection, we would like a distinction of this notion for propositions.

Definition 2.3.15. A proposition $(P, p) \colon \Omega_{\mathcal{U}}$ implies a proposition $(Q, q) \colon \Omega_{\mathcal{V}}$, written $(P, p) \Rightarrow (Q, q)$, if $P \to Q$.

Definition 2.3.16. Two propositions $P \colon \Omega_{\mathcal{U}}$ and $Q \colon \Omega_{\mathcal{V}}$ are *propositionally equivalent* $P \Leftrightarrow Q$ if $P \Rightarrow Q$ and $Q \Rightarrow P$.

With this definition, which we renamed *propositional equivalence*, we now correctly model logical equivalences for types that interpret propositions.

As with function extensionality, we can assume that this implies identification.

Definition 2.3.17. [◎] We say that *propositional extensionality* holds for a given universe $\mathcal{U}$ when, given $P, Q \colon \Omega_{\mathcal{U}}$, propositional equivalence $P \Leftrightarrow Q$ implies equality $P = Q$,

$$\mathsf{propext}(\mathcal{U}) := \prod_{(P,Q \colon \Omega_{\mathcal{U}})} (P \Leftrightarrow Q \to P = Q)$$

2.3.4 Type equivalences

Informally, and in general, two mathematical or computational structures are considered to be *equivalent* if they can be interchanged with each other in any context. Indeed, in (homotopy) type theory, two types are equivalent if their elements can be interchanged within a program without changing that program's behaviour. An early attempt at formalising this idea type-theoretically results in *quasi-inverses.*

Definition 2.3.18. [◎] Given types X and Y, a function $f \colon X \to Y$ is a *quasi-inverse* if there is a function $g \colon Y \to X$ such that either composition of f and g is the identity function:

$$\mathsf{is\text{-}qinv}(f) := \sum_{(g \colon Y \to X)} ((f \circ g \sim \mathsf{id}_Y) \times (g \circ f \sim \mathsf{id}_X)) .$$

Definition 2.3.19. Types X and Y are *quasi-invertible* if there is a quasi-inverse from X to Y:

$$\sum_{(f : X \to Y)} \mathsf{is\text{-}qinv}(f).$$

A trivial example of a quasi-inverse is the identity function, meaning that every type is quasi-invertible to itself.

While quasi-inverses indeed align with the notion of equivalence in some fields of mathematics (such as the notion of an isomorphism in category theory), there is an outstanding issue for univalent type theory: it is not the case that $\mathsf{is\text{-}qinv}(f)$ is a proposition for every f, meaning that he proof that a function is a quasi-inverse may not be unique [Uni13]. However, the statement that a function is an equivalence should be a property rather than structure.

Univalent mathematics has fixed this: there is a more general notion of equivalence that *does* yield propositions.

Definition 2.3.20. [◎] Given types X and Y, a function $f : X \to Y$ is an *equivalence* if there are functions $g, h : Y \to X$ such that $f \circ g$ and $h \circ f$ are the identity function:

$$\mathsf{is\text{-}equiv}(f) :\equiv \sum_{(g : Y \to X)} (f \circ g \sim \mathsf{id}_Y) \times \sum_{(h : Y \to X)} (h \circ f \sim \mathsf{id}_X).$$

Definition 2.3.21. [◎] Types X and Y are *equivalent* $X \simeq Y : \mathcal{U} \sqcup \mathcal{V}$ if,

$$X \simeq Y :\equiv \sum_{(f : X \to Y)} \mathsf{is\text{-}equiv}(f).$$

The fact that $\mathsf{is\text{-}qinv}(f)$ is *not* in general a proposition, while $\mathsf{is\text{-}equiv}(f)$ *is*, may at first be unintuitive. As an example of the former see Theorem 4.1.3 of [Uni13], which takes $X :\equiv \Sigma_{(A : \mathcal{U})} \|A \simeq \mathbb{2}\|$[2] and proves that $\mathsf{id}_X : X \to X$ is a quasi-inverse in two unequal ways. The (rather involved) proof of the latter follows from function extensionality — it appears as Theorem 4.3.2 of the same reference.

Of course, every quasi-inverse is trivially an equivalence. We also note here that any type that is equivalent to a proposition is a proposition.

Lemma 2.3.22. [◎] *Given $X, Y : \mathcal{U}$ such that $X \simeq Y$, if X is a proposition then so is Y.*

[2]The propositional truncation map $\| - \|$ is explained in Section 2.3.6.

Proof. Recall from Definition 2.3.5 that X being a proposition means that given any $x_1, x_2 : X$ we have $x_1 = x_2$; we wish to show the same for the equivalent type Y. Given $y_1, y_2 : Y$, we use the equivalence $f : X \to Y$ and its left-inverse $g : Y \to X$. We therefore have elements $g(y_1), g(y_2) : X$ which satisfy $g(y_1) = g(y_2)$, and therefore (by the ap function defined at the end of Section 2.2.7) also $f(g(y_1)) = f(g(y_2))$. Using the fact that $f \circ g \sim id_Y$, we can conclude $y_1 = f(g(y_1)) = f(g(y_2)) = y_2$.

When working informally, mathematicians will often conflate equivalence and equality; for example, we may say "the reals are the unique complete ordered field" and forget the qualifier "up to isomorphism". However, in MLTT, these concepts are distinct. It is obvious that equality implies equivalence:

Corollary 2.3.23. [◎] *Every type is equivalent to itself.*

Proof (Sketch). By using the identity function, which is trivially an equivalence. This proof yields the function $\simeq$-refl: $\prod_{(X : \mathcal{U})} (X \simeq X)$ in TYPETOPOLOGY.

Corollary 2.3.24. *Given two types X and Y such that $X = Y$, then $X \simeq Y$.*

Proof. We utilise the given equality $e : X = Y$ to build the function $\mathsf{transport}(\lambda(Z : \mathcal{V}).X \simeq Z, e) : X \simeq X \to X \simeq Y$. Thus, by applying Corollary 2.3.23, the result follows,

$$\begin{aligned}
&\mathsf{id\text{-}to\text{-}equiv} : \Pi_{(X,Y : \mathcal{U})} ((X = Y) \to (X \simeq Y)),\\
&\mathsf{id\text{-}to\text{-}equiv}(X, Y, e) := \mathsf{transport}(\lambda(Z : \mathcal{V}).X \simeq Z, e, \simeq\text{-}\mathsf{refl}(X)).
\end{aligned}$$

But it is not the case that equivalence implies equality — this is another independent proposition in our type theory.

2.3.5 Univalence

A foundational aim of univalent mathematics is to bring equivalence and equality into alignment. The axiom that allows us to prove that equivalence implies equality is called *univalence*. Univalence is due to Vladimir Voevodsky, who desired a foundation of mathematics in which all objects "are invariant under equivalence of structures" [ANST21].

Definition 2.3.25. [◎] We say that a given universe $\mathcal{U}$ is *univalent* when, given $X, Y : \mathcal{U}$, the function $\mathsf{id\text{-}to\text{-}equiv}(X, Y) : X = Y \to X \simeq Y$ (defined in Corollary 2.3.24) is an equivalence,

$$\text{is-univalent}(\mathcal{U}) : \prod_{(X,Y\,:\,\mathcal{U})} \text{is-equiv}(\text{id-to-equiv}(X,Y)).$$

Corollary 2.3.26. [◎] *If $\mathcal{U}$ is univalent, then all types $X, Y : \mathcal{U}$ that are equivalent $X \simeq Y$ are equal $X = Y$.*

Proof (Sketch). [u] Because, by univalence, the function $\text{id-to-equiv}(X,Y) : X = Y \to X \simeq Y$ is an equivalence, there is also a function $\text{equiv-to-id} : X \simeq Y \to X = Y$.

We use equivalences throughout this thesis, but only invoke univalence itself once (in Theorem 5.1.25). We do, however, use both function extensionality and propositional extensionality, as previously discussed.

We now state function extensionality in its non-naive form, which reflects our earlier comments on equivalence in Section 2.3.4.

Definition 2.3.27. [◎] We say that *function extensionality* holds for given universes $\mathcal{U}, \mathcal{V}$ when, given $X : \mathcal{U}$, $Y : X \to \mathcal{V}$ and functions $f, g : \prod_{(x\,:\,X)} (Y(x))$, the function $\text{happly}(f,g) : f = g \to f \sim g$ is an equivalence,

$$\text{funext}(\mathcal{U},\mathcal{V}) := \prod_{(X\,:\,\mathcal{U})} \prod_{(Y\,:\,X\to\mathcal{V})} \prod_{\left(f,g\,:\, \prod_{(x\,:\,X)}(Y(x))\right)} (\text{is-equiv}(\text{happly}(f,g)))\,.$$

Corollary 2.3.28. *If function extensionality holds for $\mathcal{U}, \mathcal{V}$ then naive function extensionality holds for $\mathcal{U}, \mathcal{V}$.*

Proof (Sketch). [f] Because, by function extensionality, the function $\text{happly}(f,g) : f = g \to f \sim g$ is an equivalence, there is also a function $\text{naive-fe} f \sim g \to f = g :$ which has the same type as naive function extensionality (Definition 2.3.4).

Univalence truly captures that equivalence and identity are equivalent: indeed, it generalises the two more specific extensionality principles.

Corollary 2.3.29. *If $\mathcal{U}$ and $\mathcal{V}$ are univalent, function extensionality holds for $\mathcal{U}, \mathcal{V}$.*

Voevodsky's original proof of this corollary is too detailed to reproduce here: we invite the interested reader to see [Gam11].

Corollary 2.3.30. *If $\mathcal{U}$ is univalent, propositional extensionality holds for $\mathcal{U}$.*

Proof (Sketch). [u] If two propositions $P, Q : \Omega_{\mathcal{U}}$ are propositionally equivalent $P \Leftrightarrow Q$, then they are immediately equivalent $P \simeq Q$. Thus, by Corollary 2.3.26 of univalence, they are equal $P = Q$.

2.3.6 Propositional truncation

Under the propositions as (some) types interpretation, we have shown the relationship between proposition types and mathematical propositions. However, with respect to the logical interpretation of the type theory, there are two outstanding problems: Σ-types represent constructive existence and $+$-types tell us exactly which of our premises holds. Both of these type families do not in general interpret propositions (see Lemmas 2.3.8 and 2.3.11) and, moreover, their interpretations as statements do not match the 'traditional' logical perspective of existence and disjunction.

This is resolved in univalent mathematics by a further axiom called *propositional truncation*, which effectively forces a structure to become a property. In TYPETOPOLOGY, propositional truncations are axiomatised in the following way.

Definition 2.3.31. [◎] We say that a given universe $\mathcal{U}$ has *propositional truncations* when there is a type truncation function $\| - \| : \mathcal{U} \to \Omega_{\mathcal{U}}$ and an element truncation function $| - | : X \to \|X\|$, such that for every function $f_X : X \to P$, where P is a proposition, there exists the truncated function $f_{\|X\|} : \|X\| \to P$.

Note that although the behaviour of the truncated function $f_{\|X\|}$ is not specified in the above axiomatisation, it indeed behaves correctly with respect to f_X and $| - |$ due to the fact that P is a proposition. By this, we mean that the commutative diagram for truncations shown in Figure 2.1 commutes.

$$\begin{tikzcd}
X \arrow[r, "|-|"] \arrow[dr, "f_X"'] & \|X\| \arrow[d, "f_{\|X\|}"] \\
 & P
\end{tikzcd}$$

Figure 2.1: Commutative diagram illustrating Lemma 2.3.32.

Lemma 2.3.32. *Given any type $X : \mathcal{U}$ in a universe which has propositional truncations and a function $f_X : X \to P$, where P is a proposition, the truncated function $f_{\|X\|} : \|X\| \to P$ is such that for all $x : X$ we have $f_X(x) = f_{\|X\|}(|x|)$.*

Proof. [t] The proof is immediate because $f_X(x)$ and $f_{\|X\|}(|x|)$ are both of type P, which is a proposition. Recall from Definition 2.3.5 that meaning that all elements of a proposition are equal.

We can truncate Σ-types to give types for interpreting traditional existence, which do not carry a constructive witness of the statement.

Definition 2.3.33. [◎] Given a type $X : \mathcal{U}$ and a type family $Y : X \to \mathcal{V}$ we define the *traditional existence type*:

$$\exists_{(x:X)} Y(x) := \| \sum_{(x:X)} Y(x) \|.$$

Furthermore, we +-types can be truncated to give traditional disjunction types, which do not label which said of their disjunction holds.

We do not use traditional disjunciton types in this thesis, though, as previously mentioned, we do use propositional truncation; in particular for traditional existence types.

2.3.7 Homotopy sets and beyond

A type $X : \mathcal{U}$ yields identities $x =_X y : \mathcal{U}$. A recognition of core importance to homotopy type theory is that $x =_X y$ *itself* yields identities $p =_{x=_X y} q : \mathcal{U}$. This means, of course, that that type also yields identities $\alpha\ {}_{p=_{x=_X y} q}\ \beta : \mathcal{U}$ — and so on ad infinitum. This recognition illuminates the concept of *homotopy-levels* (or h-levels), which assigns to some types a number that refers to how complex their yielded identity types are.

A type with *exactly* one element (a '*contractible* type') has h-level 0, a type with *at most* one element (i.e. a proposition) has h-level 1, a type whose identities have at most one element has h-level 2, a type whose identities' identities have at most one element has h-level 3, and — again — so on and so forth. This can be formally expressed by defining the following recursive function — which first requires us to formally define the notion of a contractible type.

Definition 2.3.34. [◎] A type $X : \mathcal{U}$ is *contractible* if there is some $c : X$ such that for all $x : X$ we have $c = x$:

$$\text{is-contractible } X := \Sigma_{(c : X)} \Pi_{(x : X)}\ c = x.$$

Definition 2.3.35. [◎] A type $X : \mathcal{U}$ has *h-level* $n : \mathbb{N}$ if has-h-level(X, n), defined below, holds.

$$\begin{aligned} \text{has-h-level}(X, 0) &:= \text{is-contractible } X, \\ \text{has-h-level}(X, n+1) &:= \Pi_{(x,y : X)}\ \text{has-h-level}(x =_X y, n). \end{aligned}$$

Types with h-level 2 are also called *sets* — they represent types that interpret classical sets, where elements can only be equal in one way.

Definition 2.3.36. [◎] A type $X : \mathcal{U}$ is a *set* if, for any elements $x, y : X$, we have is-prop $(x = y)$.

From there, types with h-level $n + 2$ are n-groupoids, adopting terminology from homotopy theory [Uni13].

In this thesis, we often utilise the fact that certain types are propositions or sets, but do not utilise higher types in this way.

2.4 Fundamental concepts for this thesis

To close this chapter, we introduce for reference a number of additional concepts that are used throughout the thesis.

2.4.1 Decidability and discreteness

We introduce three notions of decidability: decidability of *types*, decidability of *type-valued functions* and decidable *equality*.

Definition 2.4.1. [◎] A type is *decidable* if we can show it is either pointed or its negation is pointed:

$$\text{decidable X} := X + \neg X.$$

Definition 2.4.2. [◎] Given a type $X : \mathcal{U}$ and type-valued function $B : X \to \mathcal{V}$ the Π-type $\prod B : \mathcal{U} \sqcup \mathcal{V}$ is *decidable* (or *complemented*) if $B(a) : \mathcal{U}$ is decidable for every

$a : A$, i.e. if we have,

$$\text{complemented } B := \prod_{(a:\,A)} (\text{decidable B(a)})\,.$$

We cannot in general decide whether two elements of a type are equal; types for which we can are called *discrete*.

Definition 2.4.3. [◎] A type X is *discrete* if it has decidabile equality:

$$\text{discrete}(X) := \prod_{(x,y:\,X)} (\text{decidable } (x = y))\,.$$

The empty type, singleton type, natural numbers and integers are all discrete. Furthermore, given two discrete types X and Y, both $X + Y$ and $X \times Y$ are discrete.

2.4.2 Finite types

There are various notions of finiteness in TypeTopology, and more broadly in constructive type theory. In this thesis, we work with only those types that are *finite linearly ordered*.

Definition 2.4.4. [◎] The type family $\mathsf{Fin} : \mathbb{N} \to \mathcal{U}$ is defined by induction:

$$\begin{aligned}
&\mathsf{Fin} : \mathbb{N} \to \mathcal{U}, \\
&\mathsf{Fin}(0) && := \mathbb{0}, \\
&\mathsf{Fin}(n+1) && := \mathsf{Fin}(n) \;+\; \mathbb{1}.
\end{aligned}$$

Definition 2.4.5. [◎] A type F is *finite linearly ordered* if there is some $n : \mathbb{N}$ such that $F \simeq \mathsf{Fin}(n)$.

Note that this is structure and not property — it defines the collection of finite linear orders on F.

Definition 2.4.6. [◎] A type F is *finite* if there is some $n : \mathbb{N}$ such that $\|F \simeq \mathsf{Fin}(n)\|$.

This is now a property. Every finite linearly ordered type is trivially finite, but univalence implies that not every finite type is finite linearly ordered — hence, it is not provable to provide an order for a general finite type [Esc21].

However, in this thesis we only utilise finite linearly ordered types for a notion of finite, and so we abuse terminology and call these types 'finite' throughout.

We recall two easy lemmas concerning these finite types:

Lemma 2.4.7. [✓] *Every finite linearly ordered type F is discrete.*

Proof (Sketch.) By induction on the $n\colon \mathbb{N}$ such that $F \simeq \mathsf{Fin}(n)$. In the base case where $n := 0$, then $F \simeq \mathbb{0}$ which is vacuously discrete. In the inductive case where $n := n' + 1$, for some $n'\colon \mathbb{N}$, then $F \simeq \mathsf{Fin}(n') + \mathbb{1}$ — as both sides of this are discrete (the left-hand side by induction and right-hand side by the fact that $\mathbb{1}$ is trivially discrete), the overall $+$-type is discrete.

Lemma 2.4.8. [✓] *Every finite linearly ordered type is a set.*

Proof (Sketch.) Similarly to the above we proceed by induction and use that $\mathbb{0}$ and $\mathbb{1}$ are both sets and that the disjoint union of sets is also a set.

2.4.3 Vectors and sequences

We often require types for products beyond the binary case. For finitary products we now introduce *vectors*, and for infinitary products we introduce *sequences*. Sequences are of particular importance to this work, as the representations of real numbers we use later are sequence types.

We first give both non-dependent and dependent sequences. Given a type $X : \mathcal{U}$, the type of *(non-dependent) sequences* of elements of X is $X^{\mathbb{N}} := (\mathbb{N} \to X)$. Meanwhile, given an $\mathbb{N}$-indexed family of types $Y\colon \mathbb{N} \to \mathcal{U}$, the type of *dependent sequences* of elements of $Y(0), Y(1), Y(2), \ldots$ respectively is the Π-type $\Pi Y : \mathcal{U}$. Of course, the non-dependent version is an instance of the dependent version by setting Y as the constant function $(\lambda(-\colon \mathbb{N}).X) : \mathbb{N} \to \mathcal{U}$.

For vectors, we formally define the dependent version.

Definition 2.4.9. [◎] Given a number $n\colon \mathbb{N}$ and a finite family of types $X\colon \mathsf{Fin}(n) \to \mathcal{U}$, the type of *$n$-size (dependent) vectors* $\mathsf{Vec}(n, X)$ is defined inductively using binary products.

$$\begin{aligned}
&\mathsf{Vec} : \Pi_{(n:\,\mathbb{N})}(\mathsf{Fin}(n) \to \mathcal{U}) \to \mathcal{U}, \\
&\mathsf{Vec}(0, X) && := \mathbb{1}, \\
&\mathsf{Vec}(n+1, X) && := X_0 \times \mathsf{Vec}(n, \lambda i. X_{i+1}).
\end{aligned}$$

As with dependent sequences, each point of a dependent vector can have a different type. Again, if the given type family Y is constant on a type X, then we have defined a non-dependent vector of elements of X. We usually notate n-size non-dependent vectors as $\{x_0, ..., x_{n-1}\} : X^n$.

In the rest of the thesis we use the following (overloaded) notation for functions that operate on both vectors and sequences. Given a vector or sequence xs, we write:

- α_n for the n^{th} element of the vector/sequence,
- head xs for the first element of the vector/sequence α_0,
- tail xs for the vector/sequence with the head dropped $\lambda n. xs_{n+1}$,
- $(x :: xs)$ for the vector/sequence prepended with an element x,
- $\mathsf{map}(f, xs)$ for the vector/sequence $f(\mathsf{head}\ xs) :: \mathsf{map}(f, \mathsf{tail}\ xs)$,
- repeat x for the sequence $(\lambda n. x)$.

Note that these are not *just* notation but functions in our formalisation. For example, the following function zipWith is the extension of map to binary functions:

Definition 2.4.10. [ᚠ] Given types X, Y and Z, we define the zipWith functional that canonically lifts functions of type $(X \to Y \to Z)$ to those of type $(X^{\mathbb{N}} \to Y^{\mathbb{N}} \to Z^{\mathbb{N}})$:

$$\begin{aligned}
&\mathsf{zipWith} : (X \to Y \to Z) \to (X^{\mathbb{N}} \to Y^{\mathbb{N}} \to Z^{\mathbb{N}}), \\
&\mathsf{zipWith}(f, \alpha, \beta) := \lambda n. f(\alpha_n, \beta_n).
\end{aligned}$$

An important notion in our work is that of the *equality of sequence prefixes*, which is decidable for any length of prefix if the types of the sequence are discrete.

Definition 2.4.11. [ᚠ] Given an $\mathbb{N}$-indexed family of types $X : \mathbb{N} \to \mathcal{U}$, two sequences $\alpha, \beta : \Pi X$ have *equal n-prefixes* if they are equal at every point below n:

$$\begin{gathered}
\sim^n : \mathbb{N} \to \Pi X \to \Pi X \to \mathcal{U}, \\
\alpha \sim^n \beta := \Pi_{(i:\,\mathbb{N})}\,(i < n \to \alpha_i = \beta_i)
\end{gathered}$$

Lemma 2.4.12. [ᚠ] *Given an $\mathbb{N}$-indexed family of discrete types $X : \mathbb{N} \to \mathcal{U}$, given any sequences $\alpha, \beta : \Pi X$ we can decide whether $\alpha \sim^n \beta$ for any prefix length $n : \mathbb{N}$.*

Proof. In the base case, $\alpha \sim^0 \beta$ is always satisfied (as there is no natural number below 0). In the inductive case, we decide whether $\alpha \sim^{n+1} \beta$ by deciding whether $\alpha \sim^n \beta$ (by induction) and whether $\alpha_n = \beta_n$ (by discreteness of X_n).

Lemma 2.4.13. *Given an $\mathbb{N}$-indexed family of discrete types $X : \mathbb{N} \to \mathcal{U}$, given any sequences $\alpha, \beta : \Pi X$ the type $\alpha \sim^n \beta$, for any prefix length $n : \mathbb{N}$, is an equivalence relation (i.e. it is reflexive, symmetric and transitive).*

Proof. By those same properties (given in Section 2.2.7) on the identity type $\alpha_i = \beta_i$ for all $i < \beta$.

Chapter 3

Searchability and Continuity

In Chapter 2, we outlined our constructive and univalent foundation of mathematics that we will work within in this thesis. Within our Agda framework, we have begun to define a wide collection of mathematical structures, properties and propositions. Two structures of chief importance to this thesis are *searchability* and *total boundedness*: related concepts which are explored in this chapter in a generalised format, employed in Chapter 4 for defining convergent optimisation and regression theorems, and equipped in Chapter 6 for computation of such search, optimisation and regression procedures on types for for representing real numbers.

In order to be able to express searchability on infinite types (such as those used for representing real numbers), we need to define another key property in our framework — *continuity* on functions and predicates. In Section 3.2, we define a convenient variant of continuity on *closeness spaces*; a structure that we also explore in this chapter which also allows us to define total boundedness. By the end of this chapter, we will have two methods for constructive infinite search algorithms: by using the total boundedness property (Theorem 3.3.6) or by using the *Tychonoff theorem* for searchable types (Theorem 3.3.14). In the examples given in Chapter 6, we find that searchers derived from the former theorem are usually better suited for practical purposes, though not in every case.

3.1 Searchable Types

3.1.1 Background and motivation

Martín Escardó introduced the concept of *searchable sets* in higher computability theory [Esc08]. Informally, these sets K are those for which we can establish a computable functional $\mathcal{E}_K$, that we call a *searcher*. A searcher takes as input any Boolean-valued predicate $p : K \to \mathbb{2}$ and returns a distinguished element $\mathcal{E}_K(p) : K$ that satisfies the following *search condition*: if there is at least one element $k : K$ that satisfies p (i.e. $p(k) = 1$), then the distinguished element also satisfies p.

Infinite searchers — sometimes called 'seemingly impossible functional programs' — search elements of infinite spaces such as the type of binary sequences (i.e. elements of the *Cantor space* $\mathbb{2}^{\mathbb{N}} := \mathbb{N} \to \mathbb{2}$) [Esc12]. The searcher on the Cantor space is originally due to Berger [Ber90]. The more general searcher on the product space ΠT of infinitely-many searchable sets $T_i : \mathcal{U}$ (where $T : \mathbb{N} \to \mathcal{U}$) is due to Escardó [Esc08].

Of particular relevance to this thesis, infinite search programs have been previously defined on representations of real numbers and used to perform analysis [Esc11b]. One example is that Escardó has used infinite searchers to find solutions x to equations $f(x) = y$ [Esc13a]. Another is that Simpson has used them to perform Riemann integration and to find the global maximum value of a continuous function on the type of ternary signed-digit encodings (introduced in Section 5.2) [Sim98]. We will use searchable types for our own purposes of analysis (specifically, optimisation and regression) in Chapter 4.

3.1.2 Searchable types in MLTT

For defining searchability in our AGDA framework, there are a wealth of different options, many of which are already defined in TYPETOPOLOGY [Esc11a]. We follow Escardó's type-theoretic variants in that we have no requirement for our *type* $K : \mathcal{U}$ to be a *set* (Definition 2.3.36); further, instead of using $\mathbb{2}$ as the domain of the searched predicates, we use a type of truth values Ω. In this section, we recall definitions and proofs concerning searchable types — the proof techniques are shown only to get the reader comfortable with these techniques, which will be useful when we introduce *uniformly continuously searchable types* in Section 3.3.

In order to be able to test our searched predicates $p : K \to \Omega$, we require them to be decidable. Note that every predicate in the original formulation was decidable, because every function with domain $\mathbb{2}$ is trivially decidable — indeed, the type of decidable predicates of type $K \to \Omega$ is, by function and proposition extensionality, equivalent to the original type of predicates $K \to \mathbb{2}$.

Definition 3.1.1. [ᚠ] The type of *decidable predicates* on a type K is defined by,

$$\mathsf{decidable\text{-}predicate}(K) := \sum_{(p\colon K \to \Omega)} (\mathsf{complemented}\ p)\,.$$

We often leave the universe implicit — but note that it can be different to the universe that the co-domain type lives in — and will usually leave the witness of decidability implicit.

This gives us a definition of searchability in our type theory.

Definition 3.1.2. [◎] [ᚠ] A function $\mathcal{E}_K\colon \mathsf{decidable\text{-}predicate}(K) \to K$ is a *searcher* on a given type K if, for all $p\colon \mathsf{decidable\text{-}predicate}(K)$, it is the case that $p(\mathcal{E}_K(p))$ holds if there is some element $k\colon K$ such that $p(k)$ holds:

$$\mathsf{is\text{-}searcher}(\mathcal{E}) := \prod_{(p\colon \mathsf{decidable\text{-}predicate}(K))} \left(\sum_{(k\colon K)} p(k) \right) \to p(\mathcal{E}(p)).$$

Definition 3.1.3. [◎] [ᚠ] A type K is *searchable* if we can define a searcher on that type:

$$\mathsf{searchable}^{\mathcal{E}}(K) := \sum_{(\mathcal{E}_K\colon \mathsf{decidable\text{-}predicate}\ K \to K)} \mathsf{is\text{-}searcher}(\mathcal{E}).$$

We often use the following equivalent definition, which is more convenient:

$$\mathsf{searchable}(K) := \prod_{(p\colon \mathsf{decidable\text{-}predicate}(K))} \sum_{(k_0\colon K)} \left(\sum_{(k\colon K)} p(k) \right) \to p(k_0).$$

For non-trivial searchable types K, the searcher is not unique. As an example, consider two (informally defined) searchers for the $\mathbb{2}$ type:

1. $\mathcal{E}_{\mathbb{2}}(p) := \mathsf{if}\ p(tt)\ \mathsf{return}\ tt; \mathsf{return}\ ff;$
2. $\mathcal{E}_{\mathbb{2}}(p) := \mathsf{if}\ p(tt)\ \mathsf{return}\ ff; \mathsf{return}\ tt;$

Although both of these searchers satisfy the search condition, they can return different elements given the same predicate. Therefore, the type (searchable K) is not a subsingleton.

Note that we can always exhibit an element of a searchable type, and separately if the element returned by the searcher does not satisfy the searched predicate then no element does.

Lemma 3.1.4. [◎] [ᚠ] *Every searchable type is pointed.*

Proof. For the given searchable type K, we define the constant predicate $p^{\top}(k) :\equiv \top$, which every element satisfies. We can then introduce the element $\mathcal{E}_K(p^{\top}) : K$.

Lemma 3.1.5. [◎] *Given a searchable type K and any decidable predicate $p :$* decidable-predicate *K, if $\neg p(\mathcal{E}_K(p))$ then for all $k : K$ it is the case that $\neg p(k)$.*

Proof. Assuming $z : \neg p(\mathcal{E}_K(p))$ then given any $k : K$ we use the decidability of p to decide whether $p(k) + \neg p(k)$. In the latter case, the result follows trivially. In the former case, the search condition (in Definition 3.1.2) implies that, because there is an element satisfying the predicate, we have some $q : p(\mathcal{E}_K(p))$; therefore, this case is impossible and eliminated by $\mathbb{0}\text{-elim}(z(q))$.

3.1.3 Searching finite types

Every pointed finite linearly ordered type (as defined in Section 2.4.2) is searchable. The searcher for a finite type can simply check each element in turn (by any particular search strategy, but we assume it checks them relative to the type's linear order), returning the first element that satisfies the predicate — if the type is fully exhausted, then the searcher can return any element of the type as no such satisfying element exists.

First note that $\mathbb{1}$ is trivially searchable, and that the disjoint sum type former preserves searchability.

Remark 3.1.6. [◎] [ᚠ] $\mathbb{1}$ is searchable.

Proof. The searcher always returns $\star : \mathbb{1}$. Whether or not $p(\star)$, this satisfies the search condition.

Lemma 3.1.7. [◎] [ᚠ] *If types K and J are searchable, then so is $K + J$.*

Proof. Given any predicate $p : K + J \to \Omega$, we want to find some element $kj_0 : K + J$ that satisfies p, if such an element exists. First, we define the predicates $p_K(k) :\equiv p(\mathsf{inl}\ k)$ and $p_J(j) :\equiv p(\mathsf{inr}\ j)$.
Then, we search K to return $k_0 :\equiv \mathcal{E}_K(p_K) : K$. If $p(\mathsf{inl}\ k_0)$, then we are done and return $kj_0 :\equiv \mathsf{inl}\ k_0$. Otherwise, we go on to search J and set $j_0 :\equiv \mathcal{E}_J(p_J)$. If $p(\mathsf{inr}\ j_0)$,

then we are done and return $kj_0 := \mathsf{inl}\ k_0$; if not, then there is no element of $K + J$ that satisfies p, and so we can safely return any element.

The above allows us to show that every pointed type in the family $\mathsf{Fin} \colon \mathbb{N} \to \mathcal{U}$ is a searchable type.

Lemma 3.1.8. [ꟻ] *Given any $n \colon \mathbb{N}$, if the type $\mathsf{Fin}(n)$ is pointed then it is searchable.*

Proof. By induction on the given $n \colon \mathbb{N}$. The base case, where $n := 0$, is vacuous as $\mathsf{Fin}(0) := \mathbb{0}$ is not pointed. The inductive case, where $n := n' + 1$, requires showing $\mathsf{Fin}(n' + 1) := \mathbb{1} + \mathsf{Fin}(n')$ is searchable. As 1 is trivially searchable (see Remark 3.1.6) and $\mathsf{Fin}(n')$ is searchable by the inductive hypothesis, the result follows by Lemma 3.1.7.

We next show that any type K can be searched using a searcher on an equivalent searchable type J — i.e. equivalence preserves searchability.

Lemma 3.1.9. [ꟻ] *Given types K and J such that $K \simeq J$, if J is searchable then so is K.*

Proof. Because $K \simeq J$, we have $f \colon K \to J$, $g \colon J \to K$ and $h \colon J \to K$ such that $f \circ g \sim \mathrm{id}_J$ and $h \circ f \sim \mathrm{id}_K$.
Given any predicate $p \colon K \to \Omega$, we want to find some element $k_0 \colon K$ that satisfies p, if such an element exists. We define $p' \colon J \to \Omega$ as $p'(j) := p(g(j))$ and search J to return $j_0 := \mathcal{E}_J(p') \colon J$. If $p(g(j))$, then we are done and return $k_0 := g(j)$.
Otherwise, p' is never satisfied (i.e. $\neg p(g(j))$ for all $j \colon J$) and from this we can prove that p is never satisfied. Given $k \colon K$ such that $p(k)$, we would have $f(k) \colon J$ such that $p(k) := p(h(f(k)) := p'(f(k))$, which is a contradiction. We have proved that p is never satisfied, and hence our searcher can return any element.

We can now show that any pointed finite linearly ordered type is a searchable type.

Lemma 3.1.10. [ꟻ] *Every pointed finite linearly ordered type is searchable.*

Proof. By Lemmas 3.1.8 and 3.1.9.

Finally, we show that finite products preserve searchability.

Lemma 3.1.11. [◎] [ꟻ] *If K and J are searchable, then so is $K \times J$.*

Proof. Given any predicate $p\colon K \times J \to \Omega$, we want to find a pair of elements $(k_0, j_0)\colon K \times J$ that satisfy p, if such a pair exists. We define a family of predicates on J,

$$\begin{aligned} p_J &: K \to \text{decidable-predicate}(J), \\ p_J(k) &:= \lambda j.p(k, j). \end{aligned}$$

and a predicate on K,

$$p_K(k) := p(k, \mathcal{E}_J(p_J(k))),$$

where $\mathcal{E}_J$ is the searcher (see Definition 3.1.2) on J. By searching K for an answer to p_K, we in turn search J for an answer to $p_J(k)$ — we name the former answer $k_0\colon K$ and the latter, $\mathcal{E}_J(p_J(k_0))\colon Y$, is dependent on the former.

We now need to show that the search condition (Definition 3.1.2) is satisfied; i.e. if there is $(k, j)\colon K \times J$ such that $p(k, j)$ then also $p(k_0, \mathcal{E}_J(p_J(k_0)))$. We use the search conditions of k_0 and $\mathcal{E}_J(p_J(k_0))$:

1. If there is $k'\colon K$ such that $p_K(k') := p(k', \mathcal{E}_J(p_J(k')))$ then $p_K(k_0) := p(k_0, \mathcal{E}_J(p_J(k_0)))$,
2. For any $k^*\colon K$ if there is a $j'\colon J$ such that $p(k^*, j')$ then $p(k^*, \mathcal{E}_J(p_J(k^*)))$.

We have $p(k, j)$ and so by (2) we have $p(k, \mathcal{E}_J(p_J(k)))$, and by (1) we then have the conclusion.

3.1.4 Can we search infinite types?

Our intuition told us that all finite types are searchable, and likewise our intuition tells us that all infinite types are not searchable. Indeed, searchability of the canonical countably infinite type $\mathbb{N}$ is logically equivalent to the *limited principle of omniscience* (LPO) [TD88].

Remark 3.1.12. Recall that LPO states that, given a binary sequence, we can decide that either all points of the sequence are 0 or there is an explicit index at which point the sequence is 1.

$$\begin{aligned} &\mathsf{LPO} : \mathcal{U}, \\ &\mathsf{LPO} := \Pi_{(\alpha\colon 2^{\mathbb{N}})} \left(\Pi_{(n\colon \mathbb{N})} (\alpha_n = 0)\right) + \left(\Sigma_{(n\colon \mathbb{N})} (\alpha_n = 1)\right). \end{aligned}$$

Lemma 3.1.13. [ ] *The natural numbers are searchable if and only if LPO holds.*

Proof. We first prove that the searchability of $\mathbb{N}$ implies LPO. Given a binary sequence $\alpha : \mathbb{2}^{\mathbb{N}}$, we define the following predicate on natural numbers, which is true when the sequence at that index is 1:

$$\begin{aligned} p &: \mathbb{N} \to \Omega, \\ p(n) &:= \alpha_n = 1. \end{aligned}$$

Note that this is indeed subsingleton-valued as $\mathbb{2}$ is a set (see Lemma 2.4.8), and is decidable as $\mathbb{2}$ is discrete (see Lemma 2.4.7). Now, using the searcher $\mathcal{E}_{\mathbb{N}} :$ decidable-predicate $\mathbb{N} \to \mathbb{N}$, we obtain the element $\mathcal{E}_{\mathbb{N}}(p)$ and check whether it satisfies the predicate. If $p(\mathcal{E}_{\mathbb{N}}(p))$, then the right-hand side of LPO holds because $\alpha_{\mathcal{E}_{\mathbb{N}}(p)} = 1$. If $\neg p(\mathcal{E}_{\mathbb{N}}(p))$, then the left hand side holds because, by Lemma 3.1.5, nothing satisfies the predicate — no element of α is 1 and therefore every element must be 0).

To prove the opposite direction, we assume that LPO holds and want to show that we can find an element that satisfies any given $(p, d) :$ decidable-predicate $\mathbb{N}$ if such an element exists. Given such a predicate, we define the following binary sequence that is 1 whenever the index satisfies the predicate and 0 otherwise:

$$\begin{aligned} \alpha' &: \Pi_{(n : \mathbb{N})} \left(\mathsf{decidable}(p(n)) \to \mathbb{2}\right), \\ \alpha'(n, \mathsf{inl}\ q) &:= 1, \\ \alpha'(n, \mathsf{inr}\ z) &:= 0, \\ \alpha &: \mathbb{2}^{\mathbb{N}}, \\ \alpha_n &:= \alpha'(n, d(n)). \end{aligned}$$

The proof follows by applying LPO to α. In the case where $\alpha_n = 0$ for all $n : \mathbb{N}$, then the predicate is never satisfied and our searcher can return any element. In the case where there is some $n : \mathbb{N}$ such that $\alpha_n = 1$, then we also have a proof of $p(n)$, and the searcher can return $\mathbb{N}$. In both cases, the search condition holds.

Some infinite types, however, go against our intuitions and *are* searchable. Specifically, Martín Escardó proved that infinitary products preserve searchability; i.e. given a type family $T : \mathbb{N} \to \mathcal{U}$ of searchable types, the type $\Pi T : \mathcal{U}$ is itself searchable [Esc08]. This result is the *Tychonoff theorem* for searchable types[1] and it allows a wide range of

[1]This name comes from the relationship between the concepts of searchable types and compact spaces in (synthetic) topology. In topology, Tychonoff's theorem states that arbitrary products preserve compactness.

types, such as the aforementioned Cantor space $\mathbb{2}^{\mathbb{N}}$ and those types we use in Chapter 6 for representing compact intervals of real numbers, to be searched. We therefore require a formulation of the Tychonoff theorem in our framework for constructive type theory. However, it is not the case that we can simply translate the original proof directly into our setting.

Escardó's original infinite search algorithms were written in PCF, a language which has access to *general recursion*. The proof that this infinite search returns an answer is written in the Scott model of PCF, a setting in which all PCF-definable functions are automatically *continuous*. A corollary of this states that any decidable predicate $p : \Pi T \to \Omega$ is *uniformly continuous*, which means that only a finite amount of information about a particular search candidate $x : \Pi T$ is required to determine whether or not $p(x)$ holds. The combination of general recursion and the assumption that all functions are continuous means that the infinite search algorithm always returns an answer.

In our setting, we deliberately do not assume that all functions are continuous (in particular, this makes our mathematics compatible with classical mathematics) and we do not have access to general recursion (indeed, the recursive functions of MLTT are *primitive recursive*) [NPS90]. Therefore, we are required to use a different approach in our formulation of the Tychonoff theorem. This different approach is that we rephrase the notion of searchability to incorporate an *explicit* notion of uniform continuity. The primitive-recursive infinite search algorithms themselves are then defined using the information provided by this notion.

This results in an alternative proof of the Tychonoff theorem for searchable types, which we give in Section 3.3.3, that does not use general recursion and *explicitly* assumes that the searched predicate is uniformly continuous. Of course, this requires us first to actually define an explicit notion of continuity within our framework, which we aim to keep as general as possible for definition on a wide class of types; for this purpose, we introduce *closeness spaces* in the next section.

3.2 Closeness Spaces

Continuity, and the stronger notion of uniform continuity, are properties of functions $f : X \to Y$. Informally, continuity says that determining $f(x)$ to a requested precision ε depends on determining x to a certain precision δ, the *modulus of continuity*, which is constructed from ε and x. A *modulus of uniform continuity*, meanwhile, is constructed only from ε.

Continuity can be defined in a variety of ways; we may choose a particular definition based on the types X and Y, the form of topological information we have about those type, and/or the types of the precisions ε and δ. In this section, we motivate and define within our framework a structure we call *closeness spaces*, which exhibit a constructively economical definition of continuity and uniform continuity that we find convenient for our work.

3.2.1 Motivation via metric spaces

In analysis, one common definition of continuity is between *metric spaces*, wherein the distance between objects can be measured by a real-valued binary function called a *metric* [Kap01; Rud64]. Although we have not yet discussed what a type for real numbers looks like in our framework (this will be done later, in Chapter 5), the below definitions are for illustrative purposes only, and so the reader can assume the type of positive reals $\mathbb{R}_{\geq 0}$ in the below satisfies the expected properties.

Definition 3.2.1. [🕮] A *metric space* is a type X equipped with a *metric* $d : X \to X \to \mathbb{R}_{\geq 0}$ such that,

1. $d(x, y) = 0 \leftrightarrow x = y$,
2. $d(x, y) = d(y, x)$,
3. $d(x, z) \leq d(x, y) + d(y, z)$.

For functions $f : X \to Y$ on metric spaces, continuity says that for any $\varepsilon : \mathbb{R}_{\geq 0}$ there exists a $\delta : \mathbb{R}_{\geq 0}$ such that all elements $x, y : X$ that are δ-close to each other (i.e. $d_X(x, y) < \delta$) will be mapped to elements that are ε-close to each other (i.e. $d_Y(f(x), f(y)) < \varepsilon$). This notion of closeness can be described by the following family of reflexive and symmetric binary relations $C : \mathbb{R}_{\geq 0} \to X \to X \to \Omega$ where, for all $\varepsilon : \mathbb{R}_{\geq 0}$ and $x, y : X$, we say that x and y are ε-close if $C_\varepsilon(x, y)$.

Definition 3.2.2. [🕮] Given a metric space X, the family of *closeness relations* is defined as follows,

$$\begin{aligned} C &: \mathbb{R}_{\geq 0} \to X \to X \to \Omega, \\ C_\varepsilon(x, y) &:= d(x, y) < \varepsilon. \end{aligned}$$

Lemma 3.2.3. [🕮] *Given a metric space X and distance $\varepsilon : \mathbb{R}_{\geq 0}$, the closeness relation C_ε is reflexive and symmetric.*

Proof. Reflexivity follows from Definition 3.2.1.1; i.e. $C_\varepsilon(x, x) := (0 \leq \varepsilon)$, which is immediately satisfied for all $x : X$. Symmetry follows from Definition 3.2.1.2; i.e.

$C_\varepsilon(x, y) := (d(x, y) \leq \varepsilon) = (d(y, x) \leq \varepsilon) =: C_\varepsilon(y, x)$ for all $x, y \colon X$.

By strengthening the triangle inequality property (Definition 3.2.1.3) of metric spaces we instead define ultrametric spaces, for which closeness relations are additionally transitive and, thus, are equivalence relations.

Definition 3.2.4. [🕮] An *ultrametric space* is a type X equipped with an *ultrametric* $d : X \longrightarrow X \longrightarrow \mathbb{R}_{\geq 0}$ such that,

(i) $d(x, y) = 0 \leftrightarrow x = y$,

(ii) $d(x, y) = d(y, x)$,

(iii) $d(x, z) \leq \max(d(x, y), d(y, z))$.

Lemma 3.2.5. [🕮] *Every ultrametric space is a metric space.*

Proof. The first two conditions are the same — the third condition of ultrametric spaces implies that of metric spaces because clearly $\max(d(x, y), d(y, z)) \leq d(x, y) + d(y, z)$.

Lemma 3.2.6. [🕮] *Given an ultrametric space X and distance $\varepsilon \colon \mathbb{R}_{\geq 0}$, the closeness relation C_ε is transitive.*

Proof. Given $C_\varepsilon(x, y) := d(x, y) \leq \varepsilon$ and $C_\varepsilon(y, z) := d(y, z) \leq \varepsilon$ we have that $\max(d(x, y), d(y, z)) \leq \varepsilon$. By Definition 3.2.4.(ii), $d(x, z) \leq \max(d(x, y), d(y, z))$ and therefore by transitivity of the order $d(x, z) \leq \varepsilon =: C_\varepsilon(x, z)$.

Corollary 3.2.7. [🕮] *Given an ultrametric space X and distance $\varepsilon \colon \mathbb{R}_{\geq 0}$, the closeness relation C_ε is an equivalence relation.*

Proof. By Lemmas 3.2.3 and 3.2.6.

We now use these closeness relations to define continuity and uniform continuity on (ultra)metric spaces.

Definition 3.2.8. [🕮] Given (ultra)metric spaces X and Y, a function $f \colon X \longrightarrow Y$ is *continuous* if for all $x_1 \colon X$ and $\varepsilon \colon \mathbb{R}_{\geq 0}$ there is some $\delta \colon \mathbb{R}_{\geq 0}$ such that elements that are δ-close to x_1 map to elements that are ε-close to $f(x_1)$:

$$\begin{aligned}&\text{metric-f-continuous}(f) := \\ &\qquad \Pi_{(\varepsilon\colon \mathbb{R}_{\geq 0})}\Pi_{(x_1\colon X)}\Sigma_{(\delta\colon \mathbb{R}_{\geq 0})}\Pi_{(x_2\colon X)}\left(C_\delta(x_1,x_2) \to C_\varepsilon(f(x_1),f(x_2))\right).\end{aligned}$$

Definition 3.2.9. [🕮] Given (ultra)metric spaces X and Y, a function $f\colon X \to Y$ is *uniformly continuous* if for all $\varepsilon\colon \mathbb{R}_{\geq 0}$ there is some $\delta\colon \mathbb{R}_{\geq 0}$ such that elements that are δ-close map to elements that are ε-close:

$$\begin{aligned}&\text{metric-f-ucontinuous}(f) := \\ &\qquad \Pi_{(\varepsilon\colon \mathbb{R}_{\geq 0})}\Sigma_{(\delta\colon \mathbb{R}_{\geq 0})}\Pi_{(x_1,x_2\colon X)}\left(C_\delta(x_1,x_2) \to C_\varepsilon(f(x_1),f(x_2))\right).\end{aligned}$$

Lemma 3.2.10. [🕮] *Every uniformly continuous function between metric spaces is continuous.*

Proof. If $\delta\colon \mathbb{R}_{\geq 0}$ is a modulus of uniform continuity for $f\colon X \to Y$ (i.e. it depends only on $\varepsilon\colon \mathbb{R}_{\geq 0}$ and not on any point $x_1\colon X$) then it is a modulus of continuity for any x_1.

For the special case where the domain of the function is a truth value — and hence the function is a truth-valued predicate — the only notion of closeness we care about in the domain is logical equivalence. Therefore, we specialise uniform continuity for predicates.

Definition 3.2.11. [🕮] A predicate $p\colon X \to \Omega$ on an (ultra)metric space X is *uniformly continuous* if there is some $\delta\colon \mathbb{R}_{\geq 0}$ such that sequences that are δ-close give the same answer to the predicate:

$$\text{metric-p-ucontinuous}(p) := \Sigma_{(\delta\colon \mathbb{R}_{\geq 0})}\Pi_{(x_1,x_2\colon X)}\left(C_\delta(x_1,x_2) \to p(x_1) \Leftrightarrow p(x_2)\right).$$

3.2.2 Extended naturals and definition of closeness spaces

It will be convenient for our purposes to avoid the use of real numbers at this point and formulate, primarily for the purposes of continuity, an alternative notion of ultrametric spaces which we call *closeness spaces*. In a closeness space, the ultrametric is replace by a different binary function called a *closeness function*. The idea is that while (ultra)metrics measure the distance between the given objects, closeness functions measure the closeness between them: two identical elements of a type have closeness infinity. Thus, closeness functions give values on the type of extended natural numbers $\mathbb{N}_\infty$ defined below.

We start this subsection by recalling the TYPETOPOLOGY definition of extended naturals.

Definition 3.2.12. [◎] The *extended natural numbers* type $\mathbb{N}_\infty$ is the type of decreasing binary sequences,

$$\mathbb{N}_\infty := \sum_{(\alpha : \mathbb{2}^{\mathbb{N}})} (\text{is-decreasing } \alpha) ,$$

where $\text{is-decreasing}(\alpha) := \prod_{(i:\mathbb{N})}(\alpha_i \geq \alpha_{i+1})$.

We can lift any natural $n : \mathbb{N}$ by defining $\underline{n} : \mathbb{N}_\infty$ as the sequence of n-many 1s followed by infinitely-many 0s. $\infty : \mathbb{N}_\infty$ is therefore defined as the sequence of infinitely-many 1s. Given this, the partial order on the extended naturals — which extends that on the naturals — is given below, along with its minimum and maximum.

Definition 3.2.13. [◎] The partial order on the extended naturals is defined as follows,

$$u \preceq v := \Pi_{(n : \mathbb{N})} (u_n = 1 \to v_n = 1) .$$

Lemma 3.2.14. [◎] [◎] *The partial order on extended naturals* $\preceq : \mathbb{N}_\infty \to \mathbb{N}_\infty \to \Omega$ *is indeed a preorder (see the later Definition 4.1.4); i.e. it is reflexive and transitive.*

Proof. Both are straightforward: for reflexivity, clearly for any $n : \mathbb{N}$ if $u_n = 1$ then $u_n = 1$; for transitivity, if we have $u_n = 1 \to v_n = 1$ and also $v_n = 1 \to w_n = 1$ then $u_n = 1 \to w_n = 1$.

Lemma 3.2.15. *Given* $n, m : \mathbb{N}$ *such that* $n \leq m$*, it is the case that* $\underline{n} \preceq \underline{m}$.

Proof. Recall that $\underline{n}, \underline{m} : \mathbb{N}_\infty$ are the sequences of n-many 1s and m-many 1s respectively; i.e. $\underline{n} \sim^n 1$ and $\underline{m} \sim^m 1$. Because $n \leq m$, this means we also have $\underline{m} \sim^n 1$; thus, by transitivity of $\sim^n$ (Lemma 2.4.13) we have $\underline{n} \sim^n \underline{m}$, meaning that at every point $i : \mathbb{N}$ where $\underline{n}_i = 1$ then also $\underline{m}_i = 1$.

Lemma 3.2.16. [◎] [◎] *Given any* $u : \mathbb{N}_\infty$*, we have that* $\underline{0} \preceq u$ *and* $u \preceq \infty$.

Proof. For the former, $\underline{0}$ is never 1 and so, for every $n : \mathbb{N}$, the antecedent of the implication $\underline{0}_n = 1 \to u_n = 1$ is always empty, making the overall implication always true. For the latter, ∞ is always 1 and so the subsequent of $u_n = 1 \to \infty_n = 1$ is always true, making the overall implication always true.

In general, the partial order on the extended naturals is not decidable (i.e. we cannot decide whether or not $u \preceq v$ for any $u, v : \mathbb{N}_\infty$), but it is decidable when either side is a natural number. We give the left-hand side proof of this.

Lemma 3.2.17. [ᚠ] *Given any $n : \mathbb{N}$ and $v : \mathbb{N}_\infty$, it is the case that $\underline{n} \preceq v$ is decidable.*

Proof. We proceed by induction on $n : \mathbb{N}$. When $n := 0$, then the result follows by Lemma 3.2.16.
When $n := n' + 1$, we want to check that for every $i : \mathbb{N}$ such that $\underline{n' + 1}_i = 1$ then also $v_i = 1$. It suffices to simply check whether $v_{n'} = 1$, because we know $\underline{n' + 1}_i = 1$ for all $i \leq n'$ (by the decreasing property), and so the value of v_i only matters when $i \leq n'$; i.e. if it is 0 at any of these points then $\neg(\underline{n' + 1} \preceq v)$. By the decreasing property, if $v_i = 0$ at any point $i \leq n'$, then $v_{n'} = 0$.
We proceed by checking whether $v_{n'} = 1$ (which is decidable by Lemma 2.4.7). If $v_{n'} = 1$ then $\underline{n' + 1} \preceq v$. On the other hand, if $\neg(v_{n'} = 1)$ then $v_{n'} = 0$ and therefore $\neg(\underline{n' + 1} \preceq v)$.

We can also define the minimum of two extended naturals as the extension of the minimum of two binary digits.

Definition 3.2.18. [◎] The *minimum* of two extended naturals $\mathsf{min}(u, v) : \mathbb{N}_\infty$ is given via $\mathsf{min} := \lambda(n : \mathbb{N}).\mathsf{min}(u_n, v_n)$, which is clearly decreasing as its arguments are.

We can now define closeness spaces which, by comparing the below to Definition 3.2.4, the reader can see are a kind of dual of ultrametric spaces.

Definition 3.2.19. [ᚠ] A *closeness space* is a type X equipped with a *closeness function* $c : X \to X \to \mathbb{N}_\infty$ such that,

1. $c(x, y) = \infty \leftrightarrow x = y$,
2. $c(x, y) = c(y, x)$,
3. $\mathsf{min}(c(x, y), c(y, z)) \preceq c(x, z)$.

Remark 3.2.20 (Connection between closeness and metric spaces). Every closeness space X is informally an ultrametric space (and, hence, a metric space): using the closeness function $c\colon X \longrightarrow X \longrightarrow \mathbb{N}_\infty$, one informally defines the ultrametric $d\colon X \longrightarrow X \longrightarrow \mathbb{R}_{\geq 0}$ as $d(x, y) := 2^{-c(x,y)}$, with the usual convention that $2^{-\infty} := 0$. By this ultrametric, the distance between objects decreases towards zero as their closeness increases towards infinity.

3.2.3 Closeness relations and continuity

The reformulated definitions of closeness relations (the reader can compare to Definition 3.2.2 on metric spaces) on closeness spaces are given below.

Definition 3.2.21. [ᚠ]$^{\text{fe}}$ Given a closeness space X, the family of *closeness relations* is defined as follows,

$$\begin{aligned} C &: \mathbb{N} \to X \to X \to \Omega, \\ C_\varepsilon(x, y) &:= \underline{\varepsilon} \preceq c(x, y). \end{aligned}$$

As before, for all $\varepsilon\colon \mathbb{N}$ and $x, y\colon X$, we say that x and y are ε-close if $C_\varepsilon(x, y)$. We borrow further terminology from topology and say that if x and y are ε-close then they are in the same *ε-neighbourhood*.

Remark 3.2.22. Every element of a closeness space is 0-close to every other element by Lemma 3.2.16.

Because closeness spaces are ultrametric spaces, closeness relations on them are equivalence relations.

Lemma 3.2.23. [ᚠ] [ᚠ] [ᚠ] *Given a closeness space X and precision $\varepsilon\colon \mathbb{N}$, the closeness relation C_ε is an equivalence relation (reflexive, symmetric and transitive).*

Proof. Reflexivity follows from Definition 3.2.19.1 ; i.e. $C_\varepsilon(x, x) := \big(\underline{\varepsilon} \preceq c(x, x)\big) = \big(\underline{\varepsilon} \preceq \infty\big)$, which follows for all $x\colon X$ from Lemma 3.2.16. Symmetry follows from Definition 3.2.1.2; i.e. $C_\varepsilon(x, y) := \big(\underline{\varepsilon} \preceq c(x, y)\big) = \big(\underline{\varepsilon} \preceq c(y, x)\big) =: C_\varepsilon(y, x)$ for all $x, y\colon X$. Transitivity follows from Definition 3.2.1.3; i.e. $C_\varepsilon(x, z) := \big(\underline{\varepsilon} \preceq c(x, z)\big)$ is proved by first showing that $\big(\underline{\varepsilon} \preceq \min(c(x, y), c(y, z))\big)$, which is clearly the case by the assumptions $C_\varepsilon(x, y) := \big(\underline{\varepsilon} \preceq c(x, y)\big)$ and $C_\varepsilon(y, z) := \big(\underline{\varepsilon} \preceq c(y, z)\big)$ — the result then follows by using the transitivity of $\preceq$ (Lemma 3.2.14).

Closeness functions are monotonically decreasing: if two elements are ε_2-close, then they are ε_1-close for all $\varepsilon_1 \leq \varepsilon_2$.

Corollary 3.2.24. [Ɏ] *Given a closeness space X, precisions $\varepsilon_1, \varepsilon_2 \colon \mathbb{N}$ such that $\varepsilon_1 \leq \varepsilon_2$ and elements $x, y \colon X$, if $C_{\varepsilon_2}(x, y)$ then $C_{\varepsilon_1}(x, y)$.*

Proof. By Lemma 3.2.15.

Furthermore, we can always decide whether two elements are ε-close.

Lemma 3.2.25. [Ɏ] *Given a closeness space X, precision $\varepsilon \colon \mathbb{N}$ and elements $x, y \colon X$, it is the case that $C_\varepsilon(x, y)$ is decidable.*

Proof. By Lemma 3.2.17.

Critically, we now reformulate continuity (the reader can compare to Definitions 3.2.8, 3.2.9 and 3.2.11 on metric spaces) for closeness spaces. This definition will be used throughout the rest of the thesis.

Definition 3.2.26. [Ɏ] Given closeness spaces X and Y, a function $f \colon X \to Y$ is *continuous* if for all $x_1 \colon X$ and $\varepsilon \colon \mathbb{N}$ there is some $\delta \colon \mathbb{N}$ such that elements that are δ-close to x_1 map to elements that are ε-close to $f(x_1)$:

$$\text{f-continuous}(f) := \Pi_{(\varepsilon \colon \mathbb{N})} \Pi_{(x_1 \colon X)} \Sigma_{(\delta \colon \mathbb{N})} \Pi_{(x_2 \colon X)} \left(C_\delta(x_1, x_2) \to C_\varepsilon(f(x_1), f(x_2))\right).$$

Definition 3.2.27. [Ɏ] Given closeness spaces X and Y, a function $f \colon X \to Y$ is *uniformly continuous* if for all $\varepsilon \colon \mathbb{N}$ there is some $\delta \colon \mathbb{N}$ such that elements that are δ-close map to elements that are ε-close:

$$\text{f-ucontinuous}(f) := \Pi_{(\varepsilon \colon \mathbb{N})} \Sigma_{(\delta \colon \mathbb{N})} \Pi_{(x_1, x_2 \colon X)} \left(C_\delta(x_1, x_2) \to C_\varepsilon(f(x_1), f(x_2))\right).$$

Lemma 3.2.28. [Ɏ] *Every uniformly continuous function between closeness spaces is continuous.*

Proof. If $\delta \colon \mathbb{N}$ is a modulus of uniform continuity for $f \colon X \to Y$ (i.e. it depends only on $\varepsilon \colon \mathbb{N}$ and not on any point $x_1 \colon X$) then it is a modulus of continuity for any x_1.

Definition 3.2.29. [℣] A predicate $p\colon X \to \Omega$ on a closeness space X is *uniformly continuous* if there is some $\delta\colon \mathbb{N}$ such that sequences that are δ-close give the same answer to the predicate:

$$\text{p-ucontinuous}(p) := \Sigma_{(\delta\colon \mathbb{N})}\Pi_{(x_1,x_2\colon X)}\left(C_\delta(x_1,x_2) \to p(x_1) \Leftrightarrow p(x_2)\right).$$

As can be expected, the identity function is uniformly continuous and composition preserves uniform continuity.

Lemma 3.2.30. [℣] [℣] *The identity function is uniformly continuous and composition of functions preserves uniform continuity.*

Proof. For the identity function, we need to show for all $\varepsilon\colon \mathbb{N}$ there is $\delta\colon \mathbb{N}$ that $C_\delta(x_1,x_2)$ implies $C_\varepsilon(\mathrm{id}(x_1),\mathrm{id}(x_2))$ — thus we just set $\delta := \varepsilon$.
For composition of uniformly continuous functions $f\colon X \to Y$ and $g\colon Y \to Z$ (i.e. for all $\varepsilon\colon \mathbb{N}$ we have some $\delta_f^\varepsilon, \delta_g^\varepsilon\colon \mathbb{N}$ such that $C_{\delta_f^\varepsilon}(x_1,x_2) \to C_\varepsilon(f(x_1),f(x_2))$ and $C_{\delta_g^\varepsilon}(y_1,y_2) \to C_\varepsilon(g(y_1),g(y_2))$), we need to show that for all $\varepsilon\colon \mathbb{N}$ there is some $\delta\colon \mathbb{N}$ such that $C_\delta(x_1,x_2) \to C_\varepsilon(g(f(x_1)),g(f(x_2)))$. By setting $\delta := {\delta_f}^{{\delta_g}^\varepsilon}$, we find that $C_{{\delta_f}^{{\delta_g}^\varepsilon}}(x_1,x_2) \to C_{{\delta_g}^\varepsilon}(f(x_1),f(x_2)) \to C_\varepsilon(f(g(x_1)),f(g(x_2)))$.

Further, and importantly for the purpose of function search, we can compose uniformly continuous functions and predicates to form a new uniformly continuous predicate.

Lemma 3.2.31. [℣] *Given closeness spaces X and Y, a uniformly continuous function $f\colon X \to Y$ and a uniformly continuous (and decidable) predicate $p\colon Y \to \Omega$, the predicate*

$$p_f(x) := p(f(x)),$$

is uniformly continuous (and decidable).

Proof. The modulus of uniform continuity of the predicate $p_f\colon X \to Y$ is the modulus of uniform continuity of f at point δ, where δ is the modulus of uniform continuity of p.

Finally, we note that closeness relations themselves yield uniformly continuous and decidable predicates.

Lemma 3.2.32. [ ] [ ] *Given a closeness space X and precision $\varepsilon : \mathbb{N}$, the predicates*

$$p_l^y(x) := C_\varepsilon(x, y) \text{ and } p_r^y(x) := C_\varepsilon(y, x)$$

are uniformly continuous and decidable for any $y : X$.

Proof. By the decidability (Lemma 3.2.25) and transitivity (Lemma 3.2.23) of closeness relations.

We have shown, therefore, that closeness spaces yield a formulation of continuity that is exactly related to continuity on metric spaces; but which we can manipulate more conveniently for the use cases of this thesis.

3.2.4 Totally bounded closeness spaces

We previously mentioned that searchable types relate to compact spaces — recall from topology that a metric space is compact if it is totally bounded and complete, meaning that totally bounded spaces are generalisations of compact spaces [Sut09]. In this subsection, we will reformulate the idea of *totally bounded metric spaces* for closeness spaces, which generalise the variant of searchable types we use to search infinite types in Section 3.3.

In order to achieve this, we first define *ε-nets* on closeness spaces. Recall that an ε-net on a metric space X is a finite subset $\{x_0, ..., x_{n-1}\}$ of X such that the union of the n-many open balls with these points at their center is X itself [Sut09].

Definition 3.2.33. [🕮] Given a closeness space X, a finite linearly ordered type X' equipped with a function $g : X' \to X$ is an *ε-net of* X if for all $x : X$ there is an element $x' : X'$ such that $C_\varepsilon(x, g(x')))$.

As our interpretation of the quantifier "there is" in Definition 3.2.33 is Σ, rather than $\exists$, we can use the following equivalent definition.

Definition 3.2.34. [ ] Given a closeness space X, a finite linearly ordered type X' equipped with a function $g : X' \to X$ is an *ε-net of* X if there is a function $h : X \to X'$ such that for all $x : X$ we have $C_\varepsilon(x, g(h(x)))$.

By this definition, every element $x : X$ of a closeness space X with an ε-net (X', g) is represented by at least one point $h(x) : X'$ of the net — indeed, these points represent the whole ε-neighbourhood in which they lie. Furthermore, because closeness relations are equivalence relations for closeness spaces, two elements $x'_1, x'_2 : X'$ of the net either represent the exact same ε-neighbourhood of X or completely disjoint[2]. As an ε-net is finite, the existence of such a net implies that the number of ε-neighbourhoods of X is also finite. Because every element of a closeness space is 0-close to every other element, any pointed type (e.g. $\mathbb{1}$) is a 0-net for any closeness space).

Reflecting the definition on metric spaces, a closeness space is totally bounded if there is an ε-net for every $\varepsilon : \mathbb{N}$.

Definition 3.2.35. [ᚠ] A closeness space X is *totally bounded* if, for every precision $\varepsilon : \mathbb{N}$, there is an ε-net of X.

If a closeness space is totally bounded, it has a finite number of ε-neighbourhoods for every $\varepsilon : \mathbb{N}$.

3.2.5 Examples of (totally bounded) closeness spaces

In this subsection, we give examples of how we can build a wide class of closeness spaces, many of which are additionally totally bounded (and, therefore, admit search). Note that, unlike metric spaces, closeness spaces are only ever defined on 'intensional' mathematical spaces, such as the Cantor space $\mathbb{2}^{\mathbb{N}}$, and are not defined on 'extensional' spaces such as the unit interval $[0, 1]$.

In each case, we prove that the closeness space is indeed a closeness space (i.e. it satisfies the three properties of Definition 3.2.19). When proving a closeness function is totally bounded, the 0 case is left out due to its triviality.

Discrete closeness spaces

Definition 3.2.36. [ᚠ] Given a discrete type X, the *discrete closeness function* $c_X : X \to X \to \mathbb{N}_\infty$ is defined by case analysis of $d(x, y) : \text{decidable}(x = y)$ for arguments $x, y : X$,

- If $x = y$ then $c_X(x, y) := \infty$,
- If $x \neq y$ then $c_X(x, y) := \underline{0}$.

Lemma 3.2.37. [ᚠ] *Discrete types are closeness spaces by the discrete closeness function.*

[2]This means that each closeness space with an ε-net has a minimal ε-net obtained by discarding elements of the net that represent the same ε-neighbourhoods — though, in this thesis, we never require the net to be minimal.

Proof. We prove that each of the three conditions of Definition 3.2.19 is satisfied:

(i) (a) If $x = y$, then $c_X(x, y) := \infty$ by definition.

(b) If $c_X(x, y) := \infty$, then we check whether or not $x = y$. If $\neg(x = y)$, then we get a contradiction (because $c_X(x, y)_n := 1$ and $c_X(x, y)_n := 0$ for every $n\colon \mathbb{N}$) and hence we can derive $x = y$. Therefore, $x = y$ in both cases.

(ii) Because $x = y \leftrightarrow y = x$ and $\neg(x = y) \leftrightarrow \neg(y = x)$, $c_X(x, y) = c_X(y, x)$.

(iii) We proceed by case analysis on $d(x, y)$ and $d(y, z)$:

(a) In the case where both $x = y$ and $y = z$, then $x = z$. Hence, $\mathsf{min}(c_X(x, y), c_X(y, z)) \preceq c_X(x, z)$ reduces to $\mathsf{min}(\infty, \infty) \preceq \infty$. This is trivially satisfied because, at each point $n\colon \mathbb{N}$, it is the case that $\mathsf{min}(\infty, \infty)_n := 1$ and $\infty_n := 1$. This follows from Lemma 3.2.16.

(b) Alternatively, if either $x \neq y$ or $y \neq z$, then $\mathsf{min}(c_X(x, y), c_X(y, z)) \preceq c_X(x, z)$ reduces to $\underline{0} \preceq c_X(x, z)$. This follows from Lemma 3.2.16.

Lemma 3.2.38. [Ɏ] *Given a discrete closeness space X, if X is finite linearly ordered then it is totally bounded.*

Proof. For any $\varepsilon := \varepsilon' + 1$, the only ε-net is X itself, as for any $x\colon X$ the only element $y\colon Y$ that can satisfy $C_\varepsilon(x, y)$ is such that $y := x$; therefore $g, h\colon X \to X$ should simply be the identity map.

Disjoint union of closeness spaces

Definition 3.2.39. [Ɏ] Given closeness spaces X and Y, the *disjoint union closeness function* is defined by,

$$\begin{aligned}
&c_{X+Y} : X + Y \to X + Y \to \mathbb{N}_\infty, \\
&c_{X+Y}(\mathsf{inl}\ x_1,\ \mathsf{inl}\ x_2) && := c_X(x_1, x_2), \\
&c_{X+Y}(\mathsf{inr}\ y_1,\ \mathsf{inr}\ y_2) && := c_Y(y_1, y_2), \\
&c_{X+Y}(\mathsf{inl}\ x,\ \ \mathsf{inr}\ y) && := \underline{0}, \\
&c_{X+Y}(\mathsf{inr}\ y,\ \ \mathsf{inl}\ x) && := \underline{0}.
\end{aligned}$$

Lemma 3.2.40. [Ɏ] *Given closeness spaces X and Y, the type $X + Y$ is a closeness space by the disjoint union closeness function.*

Proof. We prove that each of the three conditions of Definition 3.2.19 is satisfied:

(i) (a) To show that $c_{X+Y}(xy_1, xy_2) = \infty$ whenever $xy_1 = xy_2$, we first recognise that the equality means either $xy_1 := \mathsf{inl}\, x_1 = \mathsf{inl}\, x_2 =: xy_2$ (for $x_1, x_2 : X$) or $xy_1 := \mathsf{inr}\, y_1 = \mathsf{inr}\, y_2 =: xy_2$ (for $y_1, y_2 : Y$) holds. As inl and inr are both embeddings (Definition 2.3.12), in the former case $x_1 = x_2$ and in the latter $y_1 = y_2$. The former case is reduced to showing $c_X(x_1, x_2) = \infty$, and the latter is reduced to showing $c_Y(y_1, y_2) = \infty$. In either case we achieve the result by the assumption that the underlying closeness functions satisfy condition (i).(a).

(b) To show that $xy_1 = xy_2$ given $c_{X+Y}(xy_1, xy_2) = \infty$, we proceed by induction on xy_1, xy_2. The latter two cases of the definition of the disjoint union closeness function (Definition 3.2.39) are impossible, as $\infty \neq \underline{0}$, and thus either $xy_1 := \mathsf{inl}\, x_1$ and $xy_2 := \mathsf{inl}\, x_2$ for $x_1, x_2 : X$ or $xy_1 := \mathsf{inr}\, y_1$ and $xy_2 := \mathsf{inr}\, y_2$ for $y_1, y_2 : X$. In the former case we now have $c_{X+Y}(\mathsf{inl}\, x_1, \mathsf{inl}\, x_2) := c_X(x_1, x_2) = \infty$ and want to show $x_1 = x_2$, and in the latter we have $c_{X+Y}(\mathsf{inr}\, y_1, \mathsf{inr}\, y_2) := c_Y(y_1, y_2) = \infty$ and want to show $y_1 = y_2$. In either case we achieve the result by the assumption that the underlying closeness functions satisfy condition (i).(b).

(ii) To show that c_{X+Y} is symmetric, we check each case of its definition (Definition 3.2.39). The first two cases are each symmetric by the assumption that the underlying closeness functions satisfy condition (ii); the latter two cases are already symmetries of each other, and so they are also symmetric.

(iii) To show that $\mathsf{min}(c_{X+Y}(xy_1, xy_2), c_{X+Y}(xy_2, xy_3)) \preceq c_{X+Y}(xy_1, xy_3)$ holds, we proceed by induction on xy_1, xy_2, xy_3:

If all three elements are from the same side of the disjoint union, the result follows by the assumption that the underlying closeness functions satisfy condition (iii),

If xy_1 and xy_2, or xy_2 and xy_3, are from different sides of the disjoint union, then the left side of the inequality reduces to $\underline{0}$ and we simply need to show $\underline{0} \preceq c_{X+Y}(xy_1, xy_3)$. This is by Lemma 3.2.16.

If xy_1 and xy_3 are from different sides of the disjoint union, then we need to show $\mathsf{min}(c_{X+Y}(xy_1, xy_2), c_{X+Y}(xy_2, xy_3)) \preceq \underline{0}$, which can only occur if $\mathsf{min}(c_{X+Y}(xy_1, xy_2), c_{X+Y}(xy_2, xy_3)) \sim \underline{0}$. This is indeed the case, as no matter which side xy_2 is from, it will differ from either xy_1 or xy_3, and so the left side of the inequality reduces to $\underline{0}$.

Lemma 3.2.41. [🐓] *Given totally bounded closeness spaces X and Y, the disjoint union closeness space $X + Y$ is totally bounded.*

Proof. For a given $\varepsilon : \mathbb{N}$, both X and Y have ε-nets; i.e. there are types X' and Y' and functions $g_X : X' \to X$, $g_Y : Y' \to Y$, $h_X : X \to X'$, $h_Y : Y \to Y'$ such that for all $x : X$ and $y : Y$ we have $C_\varepsilon(x, g_X(h_X(x)))$ and $C_\varepsilon(y, g_Y(h_Y(y)))$. To show that $X + Y$ has an ε-net, we define the following functions:

$$\begin{aligned}
&g_{X+Y} : X' + Y' \to X + Y,\\
&g_{X+Y}(\mathsf{inl}\ x') := \mathsf{inl}\ g_X(x'),\\
&g_{X+Y}(\mathsf{inr}\ y') := \mathsf{inl}\ g_Y(y'),\\
&h_{X+Y} : X + Y \to X' + Y',\\
&h_{X+Y}(\mathsf{inl}\ x) := \mathsf{inl}\ h_X(x),\\
&h_{X+Y}(\mathsf{inr}\ y) := \mathsf{inl}\ h_Y(y).
\end{aligned}$$

We then need to show that for all $xy : X + Y$ we have $C_\varepsilon(xy, g_{X+Y}(h_{X+Y}(xy)))$. This follows by induction on xy:

If $xy := \mathsf{inl}\ x$ for some $x : X$ then $C_\varepsilon(xy, g_{X+Y}(h_{X+Y}(xy)) := C_\varepsilon(x, g_X(h_X(x))$ (by definition of g_{X+Y}, h_{X+Y} and Definition 3.2.39), which we have by the fact (X', g_X) is an ε-net for X,

Similarly, if $xy := \mathsf{inr}\ y$ for some $y : Y$ then $C_\varepsilon(xy, g_{X+Y}(h_{X+Y}(xy)) := C_\varepsilon(y, g_Y(h_Y(y))$, which we have by the fact (Y', g_Y) is an ε-net for Y.

Therefore the type $X' + Y'$, equipped with the functions we defined above, is an ε-net for $X + Y$.

Finite product closeness spaces

Definition 3.2.42. [🐓] Given closeness spaces X and Y, the *binary product closeness function* is defined by,

$$\begin{aligned}
&c_{X\times Y} : X \times Y \to X \times Y \to \mathbb{N}_\infty,\\
&c_{X\times Y}((x_1, y_1), (x_2, y_2)) := \mathsf{min}(c_X(x_1, x_2), c_Y(y_1, y_2)).
\end{aligned}$$

Lemma 3.2.43. [🐓] *Given closeness spaces X and Y, the type $X \times Y$ is a closeness space by the binary product closeness function.*

Proof. [f] We prove that each of the three conditions of Definition 3.2.19 is satisfied:

(i) (a) To show that $c_{X\times Y}((x_1,y_1),(x_2,y_2)) = \infty$ whenever $(x_1,y_1) = (x_2,y_2)$, we first recognise that the equality means both $x_1 = y_1$ and $x_2 = y_2$ and therefore, by the assumption that the underlying closeness functions satisfy condition (i).(a), we have $c_X(x_1,x_2) = \infty$ and $c_Y(y_1,y_2) = \infty$. Thus, by definition of the binary product closeness function (Definition 3.2.42) we only need to show that $\mathsf{min}(\infty,\infty) = \infty$, which is immediate as $\lambda n.\mathsf{min}(1,1) = \lambda n.1$ (even without function extensionality).

(b) To show that $(x_1,y_1) = (x_2,y_2)$ given $c_{X\times Y}((x_1,y_1),(x_2,y_2)) = \infty$ then we first show that $c_X(x_1,x_2) = \infty$ and $c_Y(y_1,y_2) = \infty$ (these are because, by function extensionality, if $\mathsf{min}(u,v) = \infty$ then $u = \infty = v$). We now use the assumption that the underlying closeness functions satisfy condition (i).(b) to give us $x_1 = x_2$ and $y_1 = y_2$, and thus the result follows immediately.

(ii) The symmetry of $c_{X\times Y}$ is immediate from the assumption that the underlying closeness functions satisfy condition (ii).

(iii) We want to show that $\mathsf{min}(c_{X\times Y}((x_1,y_1),(x_2,y_2)), c_{X\times Y}((x_2,y_2),(x_3,y_3))) \preceq c_{X\times Y}((x_1,y_1),(x_3,y_3)) := \mathsf{min}(\mathsf{min}(a,c),\mathsf{min}(b,d)) \preceq \mathsf{min}(e,f)$ holds, where $a := c_X(x_1,x_2)$, $b := c_X(x_2,x_3)$, $c := c_Y(y_1,y_2)$ $d := c_Y(y_2,y_3)$, $e := c_X(x_1,x_3)$ and $f := c_Y(y_1,y_3)$. Using the assumption that the underlying closeness functions satisfy condition (iii), we have $\mathsf{min}(a,b) \preceq e$ and $\mathsf{min}(c,d) \preceq f$, and therefore $\mathsf{min}(\mathsf{min}(a,b),\mathsf{min}(c,d)) \preceq \mathsf{min}(e,f)$. The result then follows by a rearrangement of the arguments on the left-hand side of the inequality.

The following lemma, which follows easily from Definition 3.2.42, is useful when working with closeness relations of binary product closeness spaces:

Lemma 3.2.44. [ᚠ] [ᚠ] [ᚠ] *Given closeness spaces X and Y, the binary product closeness space $X \times Y$ is such that, for any $\varepsilon : \mathbb{N}$ and $(x_1,y_1),(x_2,y_2) : X \times Y$, we have*

$$C_\varepsilon((x_1,y_1),(x_2,y_2)) \Leftrightarrow C_\varepsilon(x_1,x_2) \times C_\varepsilon(y_1,y_2).$$

Proof (Sketch). This is straightforward to show once we recognise, by using the definition of the binary product closeness function (Definition 3.2.42), that we only need to show that $\underline{\varepsilon} \preceq \mathsf{min}(c_X(x_1,x_2), c_Y(y_1,y_2))$ holds if and only if both $\underline{\varepsilon} \preceq c_X(x_1,x_2)$ and $\underline{\varepsilon} \preceq c_Y(y_1,y_2)$ hold.

Lemma 3.2.45. [ᚠ] *Given totally bounded closeness spaces X and Y, the binary product closeness space $X \times Y$ is totally bounded.*

Proof. For a given $\varepsilon \colon \mathbb{N}$, both X and Y have ε-nets; i.e. there are types X' and Y' and functions $g_X \colon X' \to X$, $g_Y \colon Y' \to Y$, $h_X \colon X \to X'$, $h_Y \colon Y \to Y'$ such that for all $x \colon X$ and $y \colon Y$ we have $C_\varepsilon(x, g_X(h_X(x)))$ and $C_\varepsilon(y, g_Y(h_Y(y)))$. To show that $X \times Y$ has an ε-net, we define the following functions:

$$\begin{aligned}
g_{X\times Y} &: X' \times Y' \to X \times Y,\\
g_{X\times Y}(x', y') &:= (g_X(x'), g_Y(y')),\\
h_{X\times Y} &: X \times Y \to X' \times Y',\\
h_{X\times Y}(x, y) &:= (h_X(x), h_Y(y)).
\end{aligned}$$

We then need to show that for all $(x, y) \colon X \times Y$ we have $C_\varepsilon((x, y), g_{X\times Y}(h_{X\times Y}((x, y))))$ which, by definition of $g_{X\times Y}$, $h_{X\times Y}$ and Definition 3.2.42, reduces to $\underline{\varepsilon} \preceq \min(c_X(x, g_X(h_X(x))), c_Y(y, g_Y(h_Y(y))))$. This is by Lemma 3.2.44 and the fact that (X', g_X) and (Y', g_Y) are, respectively, ε-nets for X and Y. Therefore the type $X' \times Y'$, equipped with the functions we defined above, is an ε-net for $X + Y$.

We can use the binary product closeness function to define closeness functions for finite product types (i.e. dependent and non-dependent vectors).

Corollary 3.2.46. *Given an $(n \colon \mathbb{N})$-size vector $Y \colon \mathsf{Fin}\ n \to \mathcal{U}$ of closeness spaces, the type of n-size dependent vectors $\mathsf{Fin}\ n \to Y_n$ is a closeness space.*

Proof. [f] By induction and Lemma 3.2.43.

Corollary 3.2.47. *Given an $(n \colon \mathbb{N})$-size vector $Y \colon \mathsf{Fin}\ n \to \mathcal{U}$ of totally bounded closeness spaces, the finite product closeness space of n-size dependent vectors $\mathsf{Fin}\ n \to Y_n$ is totally bounded.*

Proof. By induction and Lemma 3.2.45.

Corollary 3.2.48. [f] [ᚠ] *Given a closeness space X, the type of n-size vectors $\mathsf{Fin}\ n \to X$ is a closeness space.*

Proof. By Corollary 3.2.46.

Corollary 3.2.49. [🐓] *Given a totally bounded closeness space X, the closeness space of n-size vectors* $\mathsf{Fin}\ n \to X$ *is totally bounded.*

Proof. By Corollary 3.2.47.

Subtype closeness functions.

Definition 3.2.50. [🐓] Given a type X, closeness space Y and a function $f : X \to Y$, the *subtype closeness function* $c_X : X \to X \to \mathbb{N}_\infty$ is defined by,

$$c_X(x_1, x_2) := c_Y(f(x_1), f(x_2)),$$

Lemma 3.2.51. [🐓] *Given a type X, closeness space Y and function $f : X \to Y$, the type X is a closeness space by the subtype closeness function if f is an embedding (Definition 2.3.12).*

Proof. Conditions 2 and 3 of Definition 3.2.19 are immediate from the fact Y is a closeness space. The same is true of one direction of the first condition (i.e. that $x = y \to c_X(x, y) = \infty$).

For the other direction of the first condition, we have $c_X(x, y) := c_Y(f(x), f(y)) = \infty$ and need to prove $x = y$. By the same condition on Y, we have $f(x) = f(y)$, and so $x = y$ follows from the fact f is an embedding.

Corollary 3.2.52. [🐓] *Given a closeness space X and truth-valued function $P : X \to \Omega$, the type $\sum_{(x : X)} (P(x))$ is a closeness space.*

Proof. By Lemma 3.2.51 because pr_1 is an embedding.

Corollary 3.2.53. [🐓] $\mathbb{N}_\infty$ *is a closeness space.*

Proof. [f] By Corollary 3.2.52 and the later Corollary 3.2.60 applied to $\mathbb{2}$.

Corollary 3.2.54. [🐓] *Given a type X and closeness space Y such that $X \simeq Y$, the type X is a closeness space.*

Proof. By Lemma 3.2.51 because the equivalence $f : X \to Y$ is an embedding.

Lemma 3.2.55. [ ] *Given a type X and totally bounded closeness space Y such that $X \simeq Y$, the equivalent closeness space X is totally bounded.*

Proof. For a given $\varepsilon \colon \mathbb{N}$, both Y has an ε-nets; i.e. there is a type Y' and functions $g_\varepsilon \colon Y' \to Y$ and $h_\varepsilon \colon Y \to Y'$ such that for all $y \colon Y$ we have $C_\varepsilon(y, g_\varepsilon(h_\varepsilon(y)))$. Furthermore, by $X \simeq Y$ there is an equivalence $f_\simeq \colon X \to Y$, and thus there is a function $g_\simeq \colon Y \to X$ such that $f_\simeq \circ g_\simeq \sim \mathsf{id}_Y$. We will now show that Y', equipped with $g := g_\simeq \circ g_\varepsilon \colon Y' \to X$, is an ε-net for X: meaning there is some function $h \colon X \to Y'$ such that for all $x \colon X$ we have $C_\varepsilon(f_\simeq(x), (f_\simeq \circ g \circ h)(x))$ (by Definition 3.2.50).

This function is defined $h := h_\varepsilon \circ f$, and we give $C_\varepsilon(f_\simeq(x), (f_\simeq \circ g_\simeq \circ g_\varepsilon \circ h_\varepsilon \circ f_\simeq)(x))$ by transitivity of C_ε (Lemma 3.2.23) after first proving (i) $C_\varepsilon(f_\simeq(x), (g_\varepsilon \circ h_\varepsilon \circ f_\simeq)(x))$ and (ii) $C_\varepsilon((g_\varepsilon \circ h_\varepsilon \circ f_\simeq)(x), (f_\simeq \circ g_\simeq \circ g_\varepsilon \circ h_\varepsilon \circ f_\simeq)(x))$:

(i) holds because (Y', g_ε) is an ε-net for Y,

(ii) holds because $f_\simeq \circ g_\simeq \sim \mathsf{id}_Y$, and therefore $(g_\varepsilon \circ h_\varepsilon \circ f_\simeq)(x) = (f_\simeq \circ g_\simeq \circ g_\varepsilon \circ h_\varepsilon \circ f_\simeq)(x)$, which in turn (by Corollary 3.2.54) yields $C_n((g_\varepsilon \circ h_\varepsilon \circ f_\simeq)(x), (f_\simeq \circ g_\simeq \circ g_\varepsilon \circ h_\varepsilon \circ f_\simeq)(x))$ for all $n \colon \mathbb{N}$.

Discrete-sequence closeness functions.

Recalling the definition of sequence prefix equality from Section 2.4.3, we define the closeness function on discrete sequence types.

Definition 3.2.56. [ ] Given an $\mathbb{N}$-indexed type family $D \colon \mathbb{N} \to \mathcal{U}$ of discrete types, the *discrete-sequence closeness function* $c_{\Pi D} \colon \Pi D \to \Pi D \to \mathbb{N}_\infty$ is defined at each point $n \colon \mathbb{N}$ by case analysis of $d(\alpha, \beta, n+1) \colon \mathsf{decidable}(\alpha \sim^n \beta)$ for arguments $\alpha, \beta \colon \Pi D$:

- If $\alpha \sim^{n+1} \beta$ then $c_{\Pi D}(\alpha, \beta)_n := 1$,
- If $\neg(\alpha \sim^{n+1} \beta)$ then $c_{\Pi D}(\alpha, \beta)_n := 0$.

This is decreasing, because $\neg(\alpha \sim^n \beta)$ implies $\neg(\alpha \sim^m \beta)$ for all $n, m \colon \mathbb{N}$ such that $m > n$.

Lemma 3.2.57. [ ] *The product of an $\mathbb{N}$-indexed type family of discrete types is a closeness space by the discrete-sequence closeness function.*

Proof (Sketch). [f] We prove each of the three conditions of Definition 3.2.19:

1. In the direction where we have $c_{\Pi D}(\alpha, \beta) = \infty$, we use the decidability of

$\alpha \sim^{n+1} \beta$ (Lemma 2.4.12) to ascertain that $\alpha \sim^{n+1} \beta$ for all $n : \mathbb{N}$ (because, if this were not the case, $c_{\Pi D}(\alpha, \beta)_n$ would equal 0). Therefore, we immediately have that $\alpha \sim \beta$ and (by function extensionality) $\alpha = \beta$. The other direction is trivial.

2. Given any $n : \mathbb{N}$, in the case where $\alpha \sim^{n+1} \beta$, by symmetry of prefix equality (Lemma 2.4.13) we also have $\beta \sim^{n+1} \alpha$, and therefore $c(\alpha, \beta)_n = c(\beta, \alpha)_n$. The same is true in the case where $\neg(\alpha \sim^{n+1} \beta)$.
3. Given any $n : \mathbb{N}$ we need to show that if $\mathsf{min}(c(\alpha, \beta)_n, c(\beta, \zeta)_n) = 1$ then $c(\alpha, \zeta)_n = 1$. The assumption means that both $c(\alpha, \beta)_n$ and $c(\beta, \zeta)_n$ are 1; hence, $\alpha \sim^{n+1} \beta$ and $\beta \sim^{n+1} \zeta$ must hold. The result then follows by transitivity of prefix equality (Lemma 2.4.13).

Lemma 3.2.58. [ᚠ] *Given an $\mathbb{N}$-indexed type family $F : \mathbb{N} \to \mathcal{U}$ of pointed finite linearly ordered types, the discrete-sequence closeness space ΠF is totally bounded.*

Proof. For any $\varepsilon : \mathbb{N}$, the closeness function considers only the ε-prefix of the sequences. Therefore, the ε-net is the type of ε-sized dependent vectors $\mathsf{Vec}(\varepsilon, F_{0,\ldots,\varepsilon-1})$, where $F_{0,\ldots,\varepsilon-1}$ is the finite-indexed type family that is the ε-prefix of F. We define the two maps for the net: $g : \mathsf{Vec}(\varepsilon,\ F_{0,\ldots,\varepsilon-1}) \to \Pi F$ is the function that outputs the sequence that is the ε-sized vector followed by repeating arbitrary elements of $F_{\varepsilon,\ldots}$ (which we attain from the pointedness of all types in F), while $h : \Pi F \to \mathsf{Vec}(\varepsilon, F_{0,\ldots,\varepsilon-1})$ is the function that gives the ε-prefix of the sequence.

Lemma 3.2.59. [ᚠ] [ᚠ] *Given an $\mathbb{N}$-indexed type family $D : \mathbb{N} \to \mathcal{U}$ of discrete types and two sequences $\alpha, \beta : \Pi D$, the types $C_n(\alpha, \beta)$ (derived from Lemma 3.2.57) and $\alpha \sim^n \beta$ are propositionally equivalent for all $n : \mathbb{N}$.*

Proof (Sketch). By induction on $n : \mathbb{N}$, though both $C_0(\alpha, \beta)$ and $\alpha \sim^0 \beta$ are trivially true, and so we only need to consider the case where $n := n' + 1$ for some $n' : \mathbb{N}$. If $C_{n+1}(\alpha, \beta) := \underline{n+1} \preceq c_{\Pi D}(\alpha, \beta)$ then clearly $c_{\Pi D}(\alpha, \beta)_n = 1$; therefore by definition of the discrete-sequence closeness function (Definition 3.2.56) and the decidability of $\sim^{n+1}$ (Lemma 2.4.12), $\alpha \sim^n \beta$ must hold. The opposite direction follows similarly, but by the decidability of C_{n+1} (Lemma 3.2.25).

Using the above, we define the non-dependent case; i.e. closeness spaces on types of sequences on a single discrete type.

Corollary 3.2.60. [ ] [ ] *The type of sequences on any finite linearly ordered type is a totally bounded closeness space.*

Proof. [f] By Lemmas 3.2.57 and 3.2.58.

Corollary 3.2.61. [ ] [ ] *Given two sequences $\alpha, \beta : \mathbb{N} \to X$ on a discrete type X, the types $C_n(\alpha, \beta)$ (derived from Corollary 3.2.60) and $\alpha \sim^n \beta$ are propositionally equivalent for all $n : \mathbb{N}$.*

Proof. By Lemma 3.2.59.

Countable product closeness spaces.

We have already shown that the product of an $\mathbb{N}$-indexed type family of discrete types is a closeness space (by Lemma 3.2.57). We will now generalise this by showing we can define a closeness function on the product of an $\mathbb{N}$-indexed type family of *closeness spaces*.

In order to do this, we will employ a diagonal argument to the countably-many extended natural values of the closeness functions within the type family $T : \mathbb{N} \to \mathcal{U}$ of closeness spaces (when applied to any given arguments $\alpha, \beta : \Pi T$). To illustrate this idea, consider Figure 3.1 which gives potential values for $c_{T_i}(\alpha_i, \beta_i)_j : \mathbb{N}_\infty$ where $i \in \{0, ..., 3\}$ and $j \in \{0, ..., 5\}$. The diagonal argument that we employ to define $c^{cs}_{\Pi T}(\alpha, \beta) : \mathbb{N}_\infty$ is such that the nth digit should be 1 if the nth diagonal is made up only of 1s. For the example, this means that $c_{\Pi T}(\alpha, \beta) := 111000....$ After the third digit, no matter the values of the other extended naturals, c_2 will always contribute a 0 to the diagonal — this shows how the result is indeed decreasing.

Extended natural \ **Index**	0	1	2	3	4	5	...
c_0	1	1	1	1	1	0	...
c_1	1	1	1	1	1	1	...
c_2	1	0	0	0	0	0	...
c_3	1	1	1	0	0	0	...
...	...	...	...	...	...	...	...

Figure 3.1: Example indices of countably-many extended naturals $c_i := c_{T_i}(\alpha_i, \beta_i)$ from countably-many closeness functions $c_{T_i} : T_i \to T_i \to \mathbb{N}_\infty$ using $\alpha, \beta : \Pi T$. The first four diagonals are coloured differently for emphasis.

This diagonalisation process can be inductively defined in our framework as follows:

Definition 3.2.62. [ ] Given an $\mathbb{N}$-indexed type family $T \colon \mathbb{N} \to \mathcal{U}$ of closeness spaces (i.e. there is a family of closeness functions $cs \colon \prod_{(n\colon \mathbb{N})} (T_n \to T_n \to \mathbb{N}_\infty)$), the *countable product closeness function* is defined by,

$$\begin{aligned}
&c^{cs}_{\Pi T} : \Pi T \to \Pi T \to \mathbb{N}_\infty,\\
&c^{cs}_{\Pi T}(\alpha, \beta)_0 && := cs_0(\alpha_0, \beta_0)_0,\\
&c^{cs}_{\Pi T}(\alpha, \beta)_{n+1} && := \mathsf{min}(cs_0(\alpha_0, \beta_0)_{n+1}, c^{\mathsf{tail}\ cs}_{\Pi(\mathsf{tail}\ T)}(\mathsf{tail}\ \alpha, \mathsf{tail}\ \beta)_n).
\end{aligned}$$

Lemma 3.2.63. [ ] *The product of an $\mathbb{N}$-indexed type family of closeness spaces is a closeness space by the countable product closeness function.*

The proof of the above is similar to Lemma 3.2.57, but requires us to use the fact that the underlying types of the type family are *themselves* closeness spaces. We recommend viewing the formalisation for more technical details on this relatively straightforward sizable proof.

Lemma 3.2.64. [ ] *Given an $\mathbb{N}$-indexed type family $T \colon \mathbb{N} \to \mathcal{U}$ of totally bounded closeness spaces, the countable product closeness space ΠT is totally bounded.*

Proof. The base case is vacuous (by Remark 3.2.22), and so we only consider the inductive case. When $\varepsilon := \varepsilon' + 1$, the ε-net is $t'_0 \times t'_s$, where t'_0 is the ε-net of T_0 and t'_s is the ε'-net of tail T by the inductive hypothesis.

Countable product closeness functions truly generalise discrete sequence closeness functions. Given a family of discrete types, one can use Lemma 3.2.37 to transform this into a family of closeness spaces. After doing this for a particular type family, the resulting discrete-sequence and countable product closeness functions will be identical.

Lemma 3.2.65. [ ] *Given an $\mathbb{N}$-indexed type family $T \colon \mathbb{N} \to \mathcal{U}$ of discrete types the discrete-sequence closeness function $c_{\Pi T}$ (Definition 3.2.56) and the countable product closeness function $c^{cs}_{\Pi T}$ (Definition 3.2.62), where $cs \colon \Pi_{(n\colon \mathbb{N})} (T_n \to T_n \to \mathbb{N}_\infty)$ is the collection of discrete closeness functions (Definition 3.2.36) on each T_n, are identical.*

Proof (Sketch). [f] We prove that the two functions are pointwise-equal — the result then follows immediately by function extensionality. To prove that, for all $\alpha, \beta \colon \Pi T$ and $n \colon \mathbb{N}$ we have $c_{\Pi T}(\alpha, \beta)_n = c^{cs}_{\Pi T}(\alpha, \beta)_n$, we proceed by induction on the given n.

In the base case where $n := 0$, we want to show that $c_{\Pi T}(\alpha, \beta)_0 = c_{T_0}(\alpha_0, \beta_0)_0$ where

$c_{T_0} := cs_0$, i.e. the discrete closeness function on T_0. Using the discreteness of T_0, we ask whether $\alpha_0 = \beta_0$. If it does then we have $\alpha \sim^1 \beta$ and therefore by definition of the discrete-sequence closeness function (Definition 3.2.56) we have $c_{\Pi T}(\alpha, \beta)_0 = 1$. We also have $c_{T_0}(\alpha_0, \beta_0) = \infty$ by definition of the discrete closeness function, and therefore clearly also $1 = c_{T_0}(\alpha_0, \beta_0)_0$. The proof technique is similar in the case where $\neg(\alpha_0 = \beta_0)$.

In the inductive case where $n := n' + 1$ for some $n' \colon \mathbb{N}$, we want to show that $c_{\Pi T}(\alpha, \beta)_n = \mathsf{min}(c_{T_0}(\alpha_0, \beta_0)_n, c_{\Pi(\mathsf{tail}\ T)}(\mathsf{tail}\ \alpha, \mathsf{tail}\ \beta)_{n'})$. Using the discreteness of each type in T, we ask whether $\alpha \sim^{n+1} \beta$. If it does, then we have $c_{\Pi T}(\alpha, \beta)_n = 1$ and therefore only need to show that (i) $c_{T_0}(\alpha_0, \beta_0)_n = 1$ and (ii) $c_{\Pi(\mathsf{tail}\ T)}(\mathsf{tail}\ \alpha, \mathsf{tail}\ \beta)_{n'} = 1$.

(i) holds by definition of the discrete closeness function, because we have shown that $\alpha_0 = \beta_0$,

(ii) holds by the inductive hypothesis once we prove that $c_{\Pi T}(\mathsf{tail}\ \alpha, \mathsf{tail}\ \beta)_{n'} = 1$; this is by definition of the discrete-sequence closeness function and the fact that $\alpha \sim^{n+1} \beta$ implies $\mathsf{tail}\ \alpha \sim^{n} \mathsf{tail}\ \beta$.

The proof technique is similar in the case where $\neg(\alpha \sim^{n+1} \beta)$.

We close this subsection by noting the following two lemmas concerning countable product closeness spaces; note that these can of course be specialised for discrete-sequence closeness spaces. The first states that if the tails of two sequences have closeness $\delta \colon \mathbb{N}$ and the head elements have closeness $\delta+1$, then the sequences themselves also have closeness $\delta + 1$.

Lemma 3.2.66. [ ] *Given an $\mathbb{N}$-indexed type family $T \colon \mathbb{N} \to \mathcal{U}$ of closeness spaces, two head elements $x_0, y_0 \colon T_0$ and two tail sequences $\alpha, \beta \colon \Pi T$ and some precision $\delta \colon \mathbb{N}$, if $C_{\delta+1}(\alpha_0, \beta_0)$ and $C_\delta(\mathsf{tail}\ \alpha, \mathsf{tail}\ \beta)$ then $C_{\delta+1}(\alpha, \beta)$.*

Proof. When $\delta := 0$, the result is vacuous by Remark 3.2.22, therefore we only consider the $\delta := \delta' + 1$ case where we need to show $\underline{\delta' + 1} \preceq c^*_{\Pi T}(\alpha, \beta)$. By the definition of the partial order on $\mathbb{N}_\infty$ (Definition 3.2.13), this is reduced to needing to show $c^*_{\Pi T}(\alpha, \beta)_{\delta'+1} = 1$.

By the definition of the closeness function on countable products (Definition 3.2.62), this means we need to show

$$\mathsf{min}(cs_0(\alpha_0, \beta_0)_{\delta'+1}, c_{\Pi(\mathsf{tail}\ T)}(\mathsf{tail}\ \alpha, \mathsf{tail}\ \beta)_{\delta'}) = 1,$$

which holds because, by our assumptions, both arguments to the min function are 1.

The second states that any sequence is infinitely close to the composition of its own head and tail — the point of this lemma is to avoid invoking function extensionality in this case.

Lemma 3.2.67. [ᚠ] *Given an* $\mathbb{N}$*-indexed type family* $T : \mathbb{N} \to \mathcal{U}$ *of closeness spaces, every sequence* $\alpha : \Pi T$ *is such that* $C_\delta(\alpha, \mathsf{head}\ \alpha :: \mathsf{tail}\ \alpha)$*, for every precision* $\delta : \mathbb{N}$*.*

Proof (Sketch). Proceeding by induction on the given δ, the base case is trivial and the inductive case is immediate by Lemma 3.2.66.

3.2.6 Pseudocloseness spaces

We have found that closeness spaces are a convenient structure to reason about the closeness of elements of a wide variety of types in our framework. Sometimes, such as for parametric regression (see Section 4.2), we find that we wish to use a more general structure so that we can reason about a wider variety of types.

We borrow the terminology of 'pseudometric spaces' and define *pseudo closeness spaces* below, in which we relax the first condition of Definition 3.2.19 such that we no longer require elements with (pseudo)closeness ∞ to be identical.

Definition 3.2.68. [ᚠ] A *pseudocloseness space* is a type X equipped with a *pseudo-closeness function* $c : X \to X \to \mathbb{N}_\infty$ such that,

1. $x = y \to c(x, y) = \infty$,
2. $c(x, y) = c(y, x)$,
3. $\mathsf{min}(c(x, y), c(y, z)) \preceq c(x, z)$.

The altered definitions of closeness relation (which remain equivalence relations) and uniform continuity for pseudocloseness spaces follow naturally from those on closeness spaces (Definitions 3.2.21, 3.2.27 and 3.2.29), and so we leave them out to avoid repetition.

The structure of a pseudocloseness space differs from a closeness space in that non-equal elements can have pseudocloseness ∞. This is required for parametric regression, for which we here define psuedocloseness spaces for function spaces, allowing functions to be compared at a finite number of given points.

Definition 3.2.69. [ᚠ] Given a type X, closeness space Y and $(n : \mathbb{N})$-size vector $xs : \mathsf{Fin}\ n \to X$ of elements of X, we define the *least-closeness pseudocloseness function* as,

$$c'_{X\to Y}{}^{(n,xs)} : (X \to Y) \to (X \to Y) \to \mathbb{N}_\infty$$
$$c'_{X\to Y}{}^{(n,xs)}(f,g) := c_{Y^n}(\mathsf{map}(f,xs), \mathsf{map}(g,xs)),$$

where $c_{Y^n} \colon (\mathsf{Fin}\ n \to X) \to (\mathsf{Fin}\ n \to X) \to \mathbb{N}_\infty$ is the closeness function derived from Corollary 3.2.48.

Least-closeness pseudocloseness functions compare two functions f and g at a finite number of given points $\{xs_0, ..., xs_{n-1}\}$, and return the minimum closeness found between these points; i.e. at each point x_i (where $i \in \{0, ..., n-1\}$) the closeness of $f(xs_i)$ and $g(xs_i)$ is computed as $c_i := c_Y(f(xs_i), g(xs_i))$, and then the minimum of these values is returned as the least-closeness pseudocloseness $c'_{X\to Y}{}^{(n,xs)}(f,g) := \mathsf{min}(c_0, ..., c_{n-1})$. This idea is inspired by least-squares approach to regression and pseudometrics on function spaces used in regression analysis [YS09].

3.3 Searching infinite types

We return, equipped with closeness spaces, to the problem of searching infinite types.

3.3.1 Uniformly continuously searchable closeness spaces

Recall from the close of Section 3.1.4 that we aim to restrict search to only those decidable predicates that have a constructive witness of their uniform continuity. These moduli of uniform continuity will be provided by closeness spaces (Definition 3.2.29).

Definition 3.3.1. [🐓] The type of *uniformly continuous and decidable predicates* on a closeness space K is defined by,

$$\mathsf{decidable\text{-}uc\text{-}predicate}(K) := \Sigma_{(p\,:\,\mathsf{decidable\text{-}predicate}\ K)}\mathsf{p\text{-}ucontinuous}\ (\mathsf{p}).$$

We now formally define the restricted definition of a searchable type (Definition 3.1.3). We call a closeness space whose uniformly continuous and decidable predicates we can search a *uniformly continuously searchable* closeness space.

Definition 3.3.2. [🐓] A function $\mathcal{E}_K \colon \mathsf{decidable\text{-}uc\text{-}predicate}(K) \to K$ is a *uniformly continuous searcher* on a given closeness space K if, for all $p \colon \mathsf{decidable\text{-}uc\text{-}predicate}(K)$, it is the case that $p(\mathcal{E}_K(p))$ holds if there is some element $k\colon K$ such that $p(k)$ holds:

$$\text{is-uc-searcher}(\mathcal{E}) := \prod_{(p\colon \text{decidable-uc-predicate}(K))} \left(\sum_{(k\colon K)} p(k) \right) \to p(\mathcal{E}(p)).$$

Definition 3.3.3. [ ] A closeness space K is *uniformly continuously searchable* if we can define a uniformly continuous searcher on that closeness space:

$$\text{uc-searchable}^{\mathcal{E}}(K) := \sum_{(\mathcal{E}_K\colon \text{decidable-uc-predicate } K \to K)} \text{is-uc-searcher}(\mathcal{E}).$$

We often use the following equivalent definition, which is more convenient:

$$\text{uc-searchable}(K) := \prod_{(p\colon \text{decidable-uc-predicate}(K))} \sum_{(k_0\colon K)} \left(\sum_{(k\colon K)} p(k) \right) \to p(k_0).$$

Remark 3.3.4. [ ] Every searchable closeness space is automatically uniformly continuously searchable by discarding the continuity information.

Much like searchable types, every uniformly continuously searchable closeness space is pointed.

Lemma 3.3.5. [ ] *Every uniformly continuously searchable closeness space is pointed.*

Proof. For the given uniformly continuously searchable space K, define the constant predicate $p^{\top}(k) := \top$, which every element satisfies and therefore has modulus of uniform continuity 0. We can then introduce the element $\mathcal{E}_K(p^{\top})\colon K$.

When searching a closeness space X, a consequence of knowing that $\delta\colon \mathbb{N}$ is a modulus of uniform continuity for the predicate $p\colon X \to \Omega$ is that instead of checking each individual candidate $x\colon X$, we can instead check each δ-neighbourhood of X collectively by a single representing element. If the representative satisfies the predicate, then we can simply return it; if not, we can discard the entire δ-neighbourhood in which it lives from the search.

A corollary to this is that if the closeness space has a finite δ-net then it has a finite number of δ-neighbourhoods and an answer to the predicate can be searched for. If the closeness space is totally bounded therefore, no matter the modulus of uniform continuity $\delta\colon \mathbb{N}$ of the predicate, it can be searched.

Theorem 3.3.6. [ ] *If a closeness space is totally bounded and pointed, then it is uniformly continuously searchable.*

Proof. Given any uniformly continuous predicate $p \colon K \to \Omega$, where K is the totally bounded closeness space to search which is pointed with $k^* : K$, we take K' to be the δ-net of K, where $\delta \colon \mathbb{N}$ is the modulus of uniform continuity of p. By definition of nets (Definition 3.2.34), K' is finite and there are functions $g \colon K' \to K$ and $h \colon K \to K'$ such that for all $k \colon K$ we have $C_\delta(k, g(h(k)))$.

As K' is finite and pointed (by $h(k^*) \colon K$), it is searchable (by Lemma 3.1.10). We therefore search it for an answer to the predicate $(p \circ g) : K' \to \Omega$, which we label $k'_0 \colon K'$. By the search condition (Definition 3.1.3), k'_0 is such that, if there is some $k' \colon K$ such that $p(g(k'))$, then $p(g(k'_0))$.

We therefore take $g(k'_0) \colon K$ as the answer to p, and must show that it satisfies the search condition; i.e. given some $k \colon K$ such that $p(k)$, it is the case that $p(g(k'_0))$. Of course, k is such that $C_\delta(x, g(h(k)))$, and therefore — by the uniform continuity of p (Definition 3.2.29) — $p(g(h(k)))$. Hence, because $(p \circ g)$ is satisfied, it is the case that $p(g(k'_0))$.

This theorem can be used to search a wide variety of infinite types (examples of which are given in the next subsection), but *cannot* be used to give a version of the Tychonoff theorem (as discussed in Section 3.1.4) in our framework — we will come back to this in Section 3.3.3.

3.3.2 Examples of uniformly continuously searchable closeness spaces

In this subsection, we give a variety of examples of uniformly continuously searchable types. Some of these are finite or totally bounded closeness spaces, while the others are preservation properties of continuous searchability that match up to those on the original definition of searchability.

Finite uniformly continuously searchable spaces

Lemma 3.3.7. [✓] *Every pointed, finite linearly ordered closeness space is uniformly continuously searchable.*

Proof. By Lemma 3.1.10 and Remark 3.3.4.

Disjoint union of uniformly continuously searchable spaces

Lemma 3.3.8. [✓] *Given uniformly continuously searchable closeness spaces K and J, the disjoint union closeness space $K + J$ is uniformly continuously searchable.*

Proof (Sketch). Given the predicate $p\colon \mathsf{decidable\text{-}predicate}(K{+}J)$, we follow the same technique as Lemma 3.1.7, except we also have to show that the predicates$p_K := p \circ \mathsf{inl}$ and $p_J := p \circ \mathsf{inr}$ are uniformly continuous by the respective closeness functions on K and J. Both of these proofs are immediate from the uniform continuity of p by the disjoint union closeness function (Definition 3.2.39), as well as the uniform continuity of inl and inr respectively, and Lemma 3.2.31.

Finite product of uniformly continuously searchable spaces

Lemma 3.3.9. [✓] *Given uniformly continuously searchable closeness spaces K and J, the binary product closeness space $K \times J$ is uniformly continuously searchable.*

Proof (Sketch.) [fp] We follow the same technique as Lemma 3.1.11, except we also have to prove that — assuming the given predicate is uniformly continuous by the binary product closeness function (Definition 3.2.42) — each predicate in the family $p_J\colon K \to \mathsf{decidable\text{-}predicate}(J)$ and the predicate $p_K\colon \mathsf{decidable\text{-}predicate}(K)$ are uniformly continuous by the respective closeness functions on J and K.
The former is straightforward by Lemma 3.2.44, while the latter uses propositional extensionality in a similar way to the later Lemma 3.3.18 but using Lemma 3.2.44 instead of Lemma 3.2.66.

Corollary 3.3.10. *Given an $n\colon \mathbb{N}$ and an $(n+1)$-size vector $Y\colon \mathsf{Fin}(n+1) \to \mathcal{U}$ of uniformly continuously searchable closeness spaces, the finite product closeness space of $(n+1)$-size dependent vectors $\mathsf{Fin}(n+1) \to Y_n$ is uniformly continuously searchable.*

Proof. [fp] By induction and Lemma 3.3.9.

Equivalent uniformly continuously searchable spaces

Lemma 3.3.11. [🐓] *Given a closeness space K and a uniformly continuously searchable closeness space J such that $K \simeq J$, the equivalent closeness space K is uniformly continuously searchable.*

Proof. Given the predicate $p\colon \text{decidable-predicate}(K)$, we follow the same technique as Lemma 3.1.9, except we also have to show that the predicate $p' := p \circ g$ (where $g\colon J \to K$ is derived from the proof of $K \simeq J$, see Definition 2.3.21) is uniformly continuous by the closeness function $c_J\colon J \to J \to \mathbb{N}_\infty$.
We first assume that $\delta\colon \mathbb{N}$ is the modulus of uniform continuity of p, which is uniformly continuous by the subtype closeness function (Definition 3.2.50) $c_K := c_J \circ f$ (where $f\colon X \to Y$ is the equivalence derived from the proof of $K \simeq J$ which is such that $f \circ g \sim \mathrm{id}_J$, again see Definition 2.3.21). Recall that this means given any $k_1, k_2\colon K$ such that $C_\delta(k_1, k_2) := \underline{\delta} \preceq c_K(k_1, k_2) := \underline{\delta} \preceq c_J(f(k_1), f(k_2))$ then $p(k_1) \Leftrightarrow p(k_2)$.
We will show that δ is also the modulus of uniform continuity for $(p \circ g)$; i.e. given $j_1, j_2\colon J$ such that $C_\delta(j_1, j_2) := \underline{\delta} \preceq c_J(j_1, j_2)$ then $p(g(j_1)) \Leftrightarrow p(g(j_2))$. Because for $i \in \{1, 2\}$ we have $j_i = f(g(j_i))$ then we also have $C_n(j_i, f(g(j_i)))$ for all $n\colon \mathbb{N}$. Therefore, by transitivity of the closeness relation (Lemma 3.2.23) we have $C_\delta(f(g(j_1)), f(g(j_2))) := \underline{\delta} \preceq c_J(f(g(j_1)), f(g(j_2)))$, and thus the result follows by the uniform continuity of p.

Finite-sequence uniformly continuously searchable spaces

One proof that discrete-sequence closeness spaces are uniformly continuously searchable comes from the fact that all such closeness spaces are totally bounded.

Corollary 3.3.12. [🐓] *Given an $\mathbb{N}$-indexed type family $F\colon \mathbb{N} \to \mathcal{U}$ of finite linearly ordered types, the discrete-sequence closeness space ΠF is uniformly continuously searchable.*

Proof. By Lemma 3.2.58 and Theorem 3.3.6.

Corollary 3.3.13. [🐓] *The type of sequences on any finite linearly ordered type is a uniformly continuously searchable closeness space.*

Proof. By Corollary 3.3.12.

However, when extracting a search algorithm from this corollary, the δ-net (where

$\delta \colon \mathbb{N}$ is the modulus of uniform continuity of the predicate being searched) must be fully computed before search can begin. This can cause efficiency issues, that are mildly improved by instead extracting a proof from the Tychonoff theorem (Theorem 3.3.14) for uniformly continuously searchable types instead[3].

3.3.3 Tychonoff theorem for uniformly continuously searchable spaces

Theorem 3.3.6 allows us to prove that the Cantor space and most of the types we wish to search in Chapter 6 are indeed uniformly continuously searchable. However, in much the same way that searchability does not imply finiteness, continuous searchability does not imply totally boundedness. This means that we cannot combine the theorem and Lemma 3.2.64 to prove that continuous searchability preserves countable products. In this subsection, we use a different proof technique, inspired by Escardó's (in [Esc08]) but which is not general recursive, to prove the Tychonoff theorem for uniformly continuously searchable spaces.

Theorem 3.3.14 (Tychonoff theorem). [ᚠ] *Given an $\mathbb{N}$-indexed type family $T \colon \mathbb{N} \to \mathcal{U}$ of uniformly continuously searchable closeness spaces, the countable product closeness space ΠT (see Definition 3.2.62) is uniformly continuously searchable.*

The proof is by induction on the searched predicate's modulus of uniform continuity $\delta \colon \mathbb{N}$. When $\delta := 0$, then the result is trivial. Otherwise, the idea of the technique is that we recursively construct finitely-many uniformly continuous and decidable predicates that test sequences $xs \colon \Pi T$ with fixed prefixes of elements. Each time the fixed prefix increases towards δ, the modulus of uniform continuity decreases towards 0; thus, the result will follow by δ-many applications of the inductive hypothesis.

We first define the following family of 'tail predicates'.

Definition 3.3.15. [ᚠ] Given an $\mathbb{N}$-indexed type family $T \colon \mathbb{N} \to \mathcal{U}$, a decidable predicate $p \colon \text{decidable-predicate}(\Pi T)$ and a fixed head element $x \colon T_0$, we define the decidable *tail predicate* as follows:

$$p_t : \text{decidable-predicate}(\Pi T) \to T_0 \to \text{decidable-predicate}(\Pi(\text{tail } T)),$$
$$p_t(p, x) := \lambda xs.p(x :: xs).$$

[3]In actuality, we rewrite the Tychonoff theorem specifically for finite-sequence spaces (as can be seen in the AGDA formalisation), but the proof method is similar, and simpler, and so we leave it out to avoid repetition.

Lemma 3.3.16. [Ɏ] *Given an $\mathbb{N}$-indexed type family $T : \mathbb{N} \to \mathcal{U}$ of uniformly continuously searchable closeness spaces and a uniformly continuous and decidable predicate $p :$ decidable-uc-predicate(ΠT) with modulus of uniform continuity $\delta + 1 : \mathbb{N}$, the tail predicate $p_t(p, x)$ for any fixed head element $x : T_0$ is uniformly continuous with modulus of uniform continuity $\delta : \mathbb{N}$.*

Proof. This follows immediately from the uniform continuity of p and Lemma 3.2.66.

This family of 'tail predicates' is matched by a family of 'head predicates', which are defined mutually recursively with the proof of Theorem 3.3.14.

Definition 3.3.17. [Ɏ] Given an $\mathbb{N}$-indexed type family $T : \mathbb{N} \to \mathcal{U}$ of uniformly continuously searchable closeness spaces and a uniformly continuous and decidable predicate $p :$ decidable-uc-predicate(ΠT) with modulus of uniform continuity $\delta + 1 : \mathbb{N}$, we define the decidable *head predicate* as follows:

$$
\begin{aligned}
&p_h : \text{decidable-uc-predicate}(\Pi T) \to \text{decidable-predicate}(T_0),\\
&p_h(p) := \lambda x. p(x :: \mathcal{E}_{\Pi(\mathsf{tail}\ T)}(p_t(p, x))),
\end{aligned}
$$

where $\mathcal{E}_{\Pi(\mathsf{tail}\ T)} :$ decidable-predicate$(\Pi(\mathsf{tail}\ T)) \to \mathsf{tail}\ T$ is the uniformly continuous searcher (see Definition 3.3.2) derived from the inductive hypothesis of Theorem 3.3.14.

Lemma 3.3.18. [Ɏ] *Given an $\mathbb{N}$-indexed type family $T : \mathbb{N} \to \mathcal{U}$ of uniformly continuously searchable closeness spaces and a uniformly continuous and decidable predicate $p :$ decidable-uc-predicate(ΠT) with modulus of uniform continuity $\delta + 1 : \mathbb{N}$, the head predicate $p_h(p, x)$ is uniformly continuous with modulus of uniform continuity $\delta + 1 : \mathbb{N}$.*

Proof. [fp] Given $x, y : T_0$, we need to show that if $C_{\delta+1}(x, y)$ then $p(x :: \mathcal{E}_{\Pi(\mathsf{tail}\ T)}(p_t(p, x)))$ implies $p(y :: \mathcal{E}_{\Pi(\mathsf{tail}\ T)}(p_t(p, y)))$. Because p is uniformly continuous with modulus of uniform continuity $\delta + 1$, this means showing that $C_\delta(x :: \mathcal{E}_{\Pi(\mathsf{tail}\ T)}(p_t(p, x)), y :: \mathcal{E}_{\Pi(\mathsf{tail}\ T)}(p_t(p, y)))$.

Because we have $C_{\delta+1}(x, y)$, we use Lemma 3.2.66 to reduce this to needing to show $C_\delta(\mathcal{E}_{\Pi(\mathsf{tail}\ T)}(p_t(p, x)), \mathcal{E}_{\Pi(\mathsf{tail}\ T)}(p_t(p, y)))$. Using propositional extensionality (Definition 2.3.16), this follows from the fact that $p_t(p, x)(xs) \Leftrightarrow p_t(p, y)(xs)$ for all $xs : \Pi X$, which we prove below.

Because $C_{\delta+1}(x, y)$ and xs is infinitely close to itself (Definition 3.2.19), then by Lemma 3.2.66 we have $C_{\delta+1}(x :: xs, y :: xs)$ and (by symmetry) $C_{\delta+1}(y :: xs, x :: xs)$. Therefore, using the uniform continuity of p, we have $p_t(p, x)(xs) \Leftrightarrow p_t(p, y)(xs)$.

To reiterate, the idea is that, given any uniformly continuous and decidable predicate p with modulus of uniform continuity $\delta + 1$, the head of the sequence x is computed by finding an answer to the head predicate $p_h(p)$, which requires finding an answer to the tail predicate $p_t(p, x)$. As the tail predicate has modulus of uniform continuity lower than the original predicate, it is recursively valid (i.e. it does not break AGDA's termination checker) to search for such an answer to the tail predicate. The process then continues until the final tail predicate has modulus of uniform continuity 0.

Proof of Theorem 3.3.14. [fp] By induction on the modulus of uniform continuity $\delta \colon \mathbb{N}$ on the predicate $p \colon \text{decidable-uc-predicate}(\Pi T)$ to be searched.

When $\delta := 0$, then by Remark 3.2.22 any element of ΠT will satisfy the predicate. Therefore, recalling Lemma 3.3.5, we return the element

$$(\lambda n. \mathcal{E}_{T_n}(p^\top)),$$

where $\mathcal{E}_{T_n} \colon \text{decidable-uc-predicate}(T_n) \longrightarrow T_n$ is the uniformly continuous searcher on T_n.

When $\delta := \delta' + 1$, then by Lemma 3.3.16 and the inductive hypothesis we construct the head predicate

$$p_h(p) := \lambda x. p(x :: \mathcal{E}_{\Pi(\mathsf{tail}\ T)}(p_t(p, x))),$$

which in turn constructs a tail predicate. By Lemma 3.3.18, this head predicate is uniformly continuous and can thus be searched because T_0 is uniformly continuously searchable — therefore, we define

$$x_0 := \mathcal{E}_{T_0}(p_h(p))).$$

We then, by Lemma 3.3.16 and the inductive hypothesis, also define

$$xs_0 := \mathcal{E}_{\Pi(\mathsf{tail}\ T)}(p_t(p, x)).$$

We return $(x_0 :: xs_0) : \Pi T$ as our answer to the predicate.

We now need to show that, if there is some $\alpha \colon \Pi T$ satisfying $p(\alpha)$, then indeed $p\ (x_0 :: xs_0)$. By Lemma 3.3.16 and the inductive hypothesis, for any $x \colon X$ the tail predicate $p_t(p, x)$ is such that if there is some $xs \colon \Pi(\mathsf{tail}\ T)$ satisfying $p_t(p, x)(xs)$ then $p_t(p, x)(\mathcal{E}_{\Pi(\mathsf{tail}\ T)}(p_t(p, x)))$. Therefore, because

$$p_t(p, \alpha_0)(\mathsf{tail}\ \alpha) := p(\alpha_0 :: \mathsf{tail}\ \alpha),$$

(by Lemma 3.2.67), we have

$$p_t(p, \alpha_0)(\mathcal{E}_{\Pi(\mathsf{tail}\ T)}(p_t(p, \alpha_0))) := p(\alpha_0 :: \mathcal{E}_{\Pi(\mathsf{tail}\ T)}(p_t(p, \alpha_0))) =: p_h(p, \alpha_0).$$

Of course, $x_0 \colon T_0$ is such that if there is some $x \colon T_0$ satisfying $p_h(p)(x)$ then $p_h(p)(x_0)$. Therefore, because we have shown α_0 satisfies $p_h(p)(\alpha_0)$, we have

$$p_h(p)(x_0) := p(x_0 :: \mathcal{E}_{\Pi(\mathsf{tail}\ T)}(p_t(p, x_0))) =: p(x_0 :: xs_0),$$

which is what we wanted.

Chapter 4

Generalised Optimisation and Regression

We are interested in how the general view of search on infinite structures yielded by uniformly continuously searchable types can be applied to the purposes of performing optimisation and regression.

These foundational concepts of analysis are usually defined on *real-valued* functions of *multiple real variables*. Informally, given a type $\mathbb{R}$ of real numbers, the central goal of *optimisation* is to compute, with mathematical guarantees, an argument which gives an approximately optimal value (e.g. a local/global minimum or maximum) of some given objective function $f : \mathbb{R}^d \to \mathbb{R}$ in the compact intervals $[a_0, b_0], ..., [a_{d-1}, b_{d-1}]$, subject to some given constraints and degree of approximation. The applications are numerous and obvious, in all areas of computational sciences, but in particular, this problem subsumes many aspects of *parametric regression* if the objective function is a loss function, representing a kind of distance between a parameterised model and a reference model (or just some sampled data), that we wish to minimise [CM02].

In this chapter, we provide a methodological contribution by describing type-theoretic variants of optimisation and regression to the general class of types (i.e. totally bounded and uniformly continuous searchable closeness spaces) we introduced in Chapter 3.

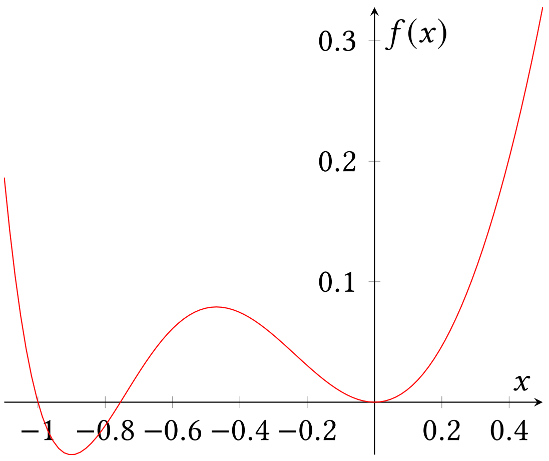


Figure 4.1: Graph of the function $f(x) = x^6 - x^4 + x^3 + x^2$ with $-1.1 \leq x \leq 1$.

4.1 Global Optimisation

Much of the recent literature in optimisation theory has focused on *local optimisation*[1] and methods for computing it efficiently, such as *gradient descent* and supporting techniques such as *automatic differentiation* [BPRS18]. Local optimisation is attractive because it can be computed efficiently, even for functions with high dimensionality.

However, there exists a mathematically attractive alternative: computing a *global minimum argument* of f in $[a, b]$, which can, for many problems, produce much stronger correctness guarantees than a local minimum [NTvD22]. A global minimum argument is an argument choice $xs\colon \mathbb{R}^d$ such that $f(xs)$ is a *global minimum value* of f — i.e., for any other choice $ys\colon \mathbb{R}^d$, we have $f(xs) \leq f(ys)$.

Remark 4.1.1. [📖] Given endpoints $a, b\colon \mathbb{R}$ such that $a \leq b$, we write $x \in [a, b]$ to mean any real $x\colon \mathbb{R}$ such that $a \leq x \leq b$.

Definition 4.1.2. [📖] Given a function $f\colon \mathbb{R} \to \mathbb{R}$ and endpoints $a, b\colon \mathbb{R}$, a given point $x_0 \in [a, b]$ is a *global minimum argument* of f in the compact interval $[a, b]$ if for all $x \in [a, b]$ we have $f(x_0) \leq f(x)$.

For example, in Figure 4.1, a global minimum argument of $f(x)$ in the compact interval $[-1.1, 1]$ is $x \approx -0.90047$ (which yields a global minimum value $f(x) \approx -0.04366$), while a local minimum is $x = 0$ (which yields a local minimum value $f(x) = 0$).

[1]Optimisation often concerns itself with finding maxima, rather than minima — however, as these two problems are equivalent (we can find a maxima of f by finding a minima of $-f$), we will only focus on finding minima.

In constructive mathematics, computing a global minimum *value* of a function on a compact interval is possible. For example, Simpson achieves this using the ternary signed-digit encoding of real numbers in the compact interval $[-1, 1]$ (explored in Section 5.2) [Sim98]. Nevertheless, the constructive existence of a global minimum *argument* — an argument to the function which gives a minimum value — is a consequence of the extreme value theorem which is not valid in constructive mathematics, meaning that they cannot in general be computed [TD88]. The fact that a global minimum argument is non-computable on the reals relies on the lack of a linear ordering: in constructive mathematics we cannot decide, for $x, y \colon \mathbb{R}$, whether $(x \leq y) + (y \leq x)$ [KK11; TD88]. Indeed, this is sometimes referred to as the analytic *lesser limited principle of omniscience* (LLPO) [Shu18].

Despite the non-computability of a global minimum argument, it is the case that, subject to continuity and compactness constraints (which are reasonable for many problem domains), an *approximate* global minimum argument x of f is computable such that $f(x)$ is a global minimum value *up to a given precision* ε.

Definition 4.1.3. [📖] Given a function $f \colon \mathbb{R} \to \mathbb{R}$, endpoints $a, b \colon \mathbb{R}$ and any precision $\varepsilon \colon \mathbb{R}$, a given point $x_0 \in [a, b]$ is an *ε-global minimum argument* of f in the compact interval $[a, b]$ if for all $x \in [a, b]$ such that $\neg C_\varepsilon(f(x_0), f(x))$ we have $f(x_0) \leq f(x)$.

Any algorithm which computes an ε-global minimum argument of a given function $f \colon \mathbb{R} \to \mathbb{R}$, for any precision $\varepsilon \colon \mathbb{R}$, is a *global optimisation algorithm*[2].

We are interested in constructing, for our wide class of types, global optimisation algorithms which are *general* (few assumptions regarding a particular shape of the function graph) and *complete* (guaranteed to find the solution within a specified margin of error). In particular, we focus on algorithms in the style of Piyavskii, which apply to continuous functions and which discretise the domain of the function similar to a branch-and-bound-style optimisation algorithm [Piy72; Kea92]. The continuity property is used for such algorithms to guarantee that the granularity of the domain can be used to bound the imprecision of the value of the objective function.

In this section, we outline how within our framework we can describe the convergent global optimisation of functions $f \colon X \to Y$, where the types X and Y are kept as general as possible (and include those types that in Chapter 5 we introduce to our type theory in order to encode the real numbers).

[2] Global optimisation algorithms in general are defined for functions of multiple real variables; however, as our generalised perspective will eliminate the difference between the two, we stated only the single argument version for ease of presentation.

4.1.1 Orders and approximate orders

In order to reason about the minima of functions $f\colon X \to Y$, our theory must have a concept of an order on those value types Y. We remain general in our idea of exactly which types these are, as we wish to establish a theory of *generalised* optimisation and regression.

We first recall some basic notions of orders on a given type.

Definition 4.1.4. [ ] For any type X, a binary relation $\leq\colon X \to X \to \Omega$ is a *preorder on* X if it is reflexive and transitive:

(i) $x \leq x$,

(ii) $x \leq y \to y \leq z \to x \leq z$.

Definition 4.1.5. [ ] For any type X, a binary relation $\leq\colon X \to X \to \Omega$ is a *linear preorder on* X if it is a preorder and if, for all $x, y\colon X$, it is the case that $(x \leq y) + (y \leq x)$ holds.

Definition 4.1.6. [ ] For any type X, a binary relation $\leq\colon X \to X \to \Omega$ is a *partial order on* X if it is an antisymmetric preorder; i.e. if for any $x, y\colon X$ such that $x \leq y$ and $y \leq x$ it is the case that $x = y$.

Definition 4.1.7. [ ] For any type X, a binary relation $\leq\colon X \to X \to \Omega$ is a *linear order on* X if it is a preorder that is both antisymmetric and linear.

The easiest example of a class of types with linear orders are the finite linearly ordered types.

Remark 4.1.8. [ ] Every finite linearly ordered type F has a linear order $\leq_F\colon F \to F \to \Omega$ inherited from $\mathsf{Fin}(n)$ as discussed in Section 2.4.2.

Recall that in the preamble to this section, we noted that we cannot compute a global minimum argument of a function $f\colon \mathbb{R} \to \mathbb{R}$ because we cannot prove constructively that the expected preorder on the reals is linear. We illustrate this further, without using the reals, by the following motivating example of a class of infinite types $D^{\mathbb{N}}$ (where D is a discrete set) which has a partial order $\leq_D\colon D \to D \to \Omega$ whose linearity — similarly to that of the reals — amounts (in non-trivial cases) to a constructive taboo. This example is not just chosen for its intelligibility; this class of types is crucial for our later purposes of representing computable real numbers.

We first define the usual *lexicographic order* on the discrete-sequence type $D^{\mathbb{N}}$ —

recall from the literature that this order says $\alpha \leq_{D^{\mathbb{N}}} \beta$ if at every point $n \colon \mathbb{N}$ that α and β agree prior to, we have $\alpha_n \leq_{seqD} \beta_n$. For example, setting $D := \mathbb{2}$,

$$\{0,0,0,1,0,\ldots\} \leq_{seqD} \{0,0,0,1,1,\ldots\} \leq_{seqD} \{0,1,0,1,1,\ldots\} \leq_{seqD} \{1,1,1,1,1,\ldots\}.$$

Definition 4.1.9. [℣] Given a preorder $\leq_D \colon D \to D \to \Omega$ on a discrete set D, the *lexicographic order* $\leq_{D^{\mathbb{N}}} \colon D^{\mathbb{N}} \to D^{\mathbb{N}} \to \Omega$ on $D^{\mathbb{N}}$ is defined:

$$\begin{aligned} \leq_{D^{\mathbb{N}}} \quad &\colon D^{\mathbb{N}} \to D^{\mathbb{N}} \to \Omega, \\ \alpha \leq_{D^{\mathbb{N}}} \beta &:= \Pi_{(n \colon \mathbb{N})} \left(\alpha \sim^n \beta \to \alpha_n \leq_D \beta_n\right). \end{aligned}$$

Lemma 4.1.10. [℣] *Given a partial order $\leq_D \colon D \to D \to \Omega$ on a discrete set D, the lexicographic order $\leq_{D^{\mathbb{N}}} \colon D^{\mathbb{N}} \to D^{\mathbb{N}} \to \Omega$ is a preorder on $D^{\mathbb{N}}$.*

Proof. [f] We prove both reflexivity and transitivity of the lexicographic order:

(i) Given $\alpha \colon D^{\mathbb{N}}$, then $\alpha \leq_{D^{\mathbb{N}}} \alpha$ because, by reflexivity of the partial order on D, $\alpha_n \leq_D \alpha_n$ for all $n \colon \mathbb{N}$.

(ii) Given $\alpha, \beta, \zeta \colon D^{\mathbb{N}}$, such that $\alpha \leq_{D^{\mathbb{N}}} \beta$ and $\beta \leq_{D^{\mathbb{N}}} \zeta$ we need to show that, given any $n \colon \mathbb{N}$ such that $\alpha \sim^n \zeta$ we have $\alpha_n \leq_D \zeta_n$. We continue by induction on the given n.

In the case where $n := 0$ then we have $\alpha_0 \leq_D \beta_0$ and $\beta_0 \leq_D \zeta_0$, because $\alpha \sim^0 \beta$ and $\beta \sim^0 \zeta$ trivially hold; therefore $\alpha_0 \leq_D \zeta_0$ holds by transitivity of the partial order on D.

In the case where $n := n' + 1$ then we have $\alpha \sim^{n'+1} \zeta$ (and hence tail $\alpha \sim^{n'}$ tail ζ) and want to show that (tail $\alpha)_{n'} \leq_D$ (tail $\zeta)_{n'}$. We do this by invoking the inductive hypothesis after first proving tail $\alpha \leq_{D^{\mathbb{N}}}$ tail β and tail $\beta \leq_{D^{\mathbb{N}}}$ tail ζ.

In order to prove that tail $\alpha \leq_{D^{\mathbb{N}}}$ tail β, we need to show that, given any $i \colon \mathbb{N}$ such that tail $\alpha \sim^i$ tail β we have (tail *alpha*$)_i \leq_D$ (tail $\beta)_i$. This follows from the fact that $\alpha \leq_{D^{\mathbb{N}}} \beta$ once we show that $\alpha \sim^{i+1} \beta$; i.e. for all $j \colon \mathbb{N}$ such that $j < i + 1$ we have $\alpha_j = \beta_j$. This is shown by induction on j.

In the case where $j := 0$, we have both $\alpha_0 \leq_D \beta_0$ (trivially from $\alpha \leq_{D^{\mathbb{N}}} \beta$) and $\beta_0 \leq_D \alpha_0$ (by $\alpha \sim^{n'+1} \zeta$ and $\beta \leq_{D^{\mathbb{N}}} \zeta$). Therefore $\alpha_0 = \beta_0$ follows by antisymmetry of the partial order on D.

In the case where $j := j' + 1$ then we have $j' < i$; therefore $\alpha_{j'+1} = \beta_{j'+1}$ follows immediately from tail $\alpha \sim^i$ tail β.

The proof that tail $\beta \leq_{D^{\mathbb{N}}}$ tail ζ is by the same reasoning.

Assuming this order is linear allows us to prove a constructive taboo, the *weak principle of omniscience* (WLPO) — therefore, the linearity of this order is constructively invalid.

Remark 4.1.11. WLPO states that, given a binary sequence, we can decide whether or not all points of the sequence are 1. Another way of saying this is that, given any extended natural number, we can decide whether or not it is equal to infinity.

$$\begin{aligned} &\mathsf{WLPO} : \mathcal{U}, \\ &\mathsf{WLPO} := \Pi_{(u\,:\,\mathbb{N}_\infty)} \left(\text{is-decidable}(u = \infty)\right). \end{aligned}$$

Note that WLPO is implied by LPO (Remark 3.1.12). Martín Escardó has previously shown that WLPO is equivalent to the existence of a function $f\colon \mathbb{N}_\infty \to \mathbb{2}$ that is *discontinuous* in the sense that, given some $u\colon \mathbb{N}_\infty$, it is the case that $f(u) = 0$ if $u = \overline{n}$ (for some $n\colon \mathbb{N}$) but $f(u) = 1$ if $u = \infty$ [ES16].

$$\begin{aligned} &\mathsf{WLPO}' : \mathcal{U}, \\ &\mathsf{WLPO}' := \\ &\qquad \Sigma_{(f\,:\,\mathbb{N}_\infty \to \mathbb{2})} \Pi_{(u\,:\,\mathbb{N}_\infty)} \left(\left(\Sigma_{(n\,:\,\mathbb{N})}\,(u = \overline{n}) \to f(u) = 0\right) \times (u = \infty \to f(u) = 1)\right). \end{aligned}$$

Lemma 4.1.12. [◎] [Ɏ] *The linearity of the lexicographic order $\leq_{F^{\mathbb{N}}}\colon F^{\mathbb{N}} \to F^{\mathbb{N}} \to \Omega$ on $F^{\mathbb{N}}$, where F is a finite linearly ordered type with more than one element, is logically equivalent to WLPO.*

Proof. The first, rather straightforward step, is showing that the linearity of $\leq_{F^{\mathbb{N}}}\colon F^{\mathbb{N}} \to F^{\mathbb{N}} \to \Omega$ implies the linearity of $\preceq\colon \mathbb{N}_\infty \to \mathbb{N}_\infty \to \Omega$ (as defined in Definition 3.2.13). The idea here is that, because F is finite linearly ordered and has more than one element, we define an order-preserving function $\rho\colon \mathbb{2} \to F$ — i.e. for $a, b\colon \mathbb{2}$ if $a \leq_{\mathbb{2}} b$ then $\rho(a) \leq_F \rho(b)$ — that we use pointwise to define an order-preserving function $\mathsf{map}(\rho)\colon \mathbb{2}^{\mathbb{N}} \to F^{\mathbb{N}}$. This means that, given any $u, v\colon \mathbb{N}_\infty$, it is the case that $u \preceq v$ holds if and only if $\mathsf{map}(\rho, \mathsf{fst}(u)) \leq_{F^{\mathbb{N}}} \mathsf{map}(\rho, \mathsf{fst}(v))$ holds. Using this, the linearity of $\leq_{F^{\mathbb{N}}}$ implies the linearity of $\preceq$.
We can now proceed with the core of the proof[a], i.e. that the linearity of $\preceq$ implies WLPO′ (as given in Remark 4.1.11). Using the linearity of $\preceq$, we define a function linearity-decider: $\mathbb{N}_\infty \to \mathbb{N}_\infty \to \mathbb{2}$ that on input of two extended naturals $u, v\colon \mathbb{N}_\infty$

checks which side of $(u \preceq v) + (v \preceq u)$ holds and returns 0 if the left-hand side holds and 1 if the right-hand side holds. Of course, given two identical items either 0 or 1 can be returned. We make the following three observations for all $n : \mathbb{N}$:

1. $\mathsf{linearity\text{-}decider}(\overline{n}, \infty) = 0$,
2. $\mathsf{linearity\text{-}decider}(\infty, \overline{n}) = 1$,
3. $\mathsf{linearity\text{-}decider}(\infty, \infty)$ could be either 0 or 1.

Using these observations and the function, we define a discontinuous function $f : \mathbb{N}_\infty \to \mathbb{2}$ that returns 0 on a finite input and 1 on an infinite input. We do this by comparing ∞ to *itself* and defining f based on what is output.

If $\mathsf{linearity\text{-}decider}(\infty, \infty) = 1$ then we define $f(x) := \mathsf{linearity\text{-}decider}(x, \infty)$. On finite input (when $x = \overline{n}$ for some $n : \mathbb{N}$), it is the case that $f(x) := \mathsf{linearity\text{-}decider}(\overline{n}, \infty) = 0$ (by the first observation above). On infinite input (when $x = \infty$) then it is the case that $f(x) := \mathsf{linearity\text{-}decider}(\infty, \infty) = 1$.

If $\mathsf{linearity\text{-}decider}(\infty, \infty) = 0$ then we define $f(x) := \mathsf{flip}_{\mathbb{2}}(\mathsf{linearity\text{-}decider}(\infty, x))$. On finite input, it is the case that $f(x) := \mathsf{flip}_{\mathbb{2}}(\mathsf{linearity\text{-}decider}(\infty, \overline{n})) = \mathsf{flip}_{\mathbb{2}}(1) = 0$ (by the second observation above). On infinite input (when $x = \infty$) then it is the case that $f(x) := \mathsf{flip}_{\mathbb{2}}(\mathsf{linearity\text{-}decider}(\infty, \infty)) = \mathsf{flip}_{\mathbb{2}}(0) = 1$.

In both cases we have defined a discontinuous function, and therefore the linearity of $\preceq$ allows us to define WLPO′ and (by Remark 4.1.11) WLPO itself.

[a]This interesting proof was written and shown to me by Martín Escardó while I was completing my thesis corrections, who gave me permission to reproduce it here.

A global optimisation algorithm must compute an approximate global minimum argument; in order to do this, we require some form of approximate linear preordering that relates to the underlying pre-order we are unable to prove the linearity of. We therefore use our closeness spaces to define this concept of approximate linear preorders.

Definition 4.1.13. [ᚠ] For any closeness space X, an $\mathbb{N}$-indexed family of binary relations $\preceq^{-} : \mathbb{N} \to X \to X \to \Omega$ is an *approximate linear preorder* if, for any $\varepsilon : \mathbb{N}$, the relation $\preceq^{\varepsilon} : X \to X \to \Omega$ is a linear preorder satisfying,

(i) $\mathsf{decidable}(x \preceq^{\varepsilon} y)$,
(ii) $C_\varepsilon(x, y) \to x \preceq^{\varepsilon} y$,

The idea of the above definition is that two elements of a closeness space with an approximate linear preorder can be ordered in a decidable way and that ε-close elements must be considered indistinguishable from the point of view of the approximate order.

In the following definition, we state what it means for an approximate linear preorder to relate to the *genuine* preorder.

Definition 4.1.14. [Ƴ] For any closeness space X, an approximate linear preorder $\leq^{-} : \mathbb{N} \to X \to X \to \Omega$ *relates to* a preorder $\leq : X \to X \to \Omega$ if the following hold:

(i) $x \leq y \to \exists_{(n:\,\mathbb{N})}\Pi_{(\varepsilon:\,\mathbb{N})}\,(n < \varepsilon \to x \leq^{\varepsilon} y)$,

(ii) $\Pi_{(n:\,\mathbb{N})}\,(x \leq^{n} y) \to x \leq y$

This first half of the above relationship says that if the two elements are genuinely such that $x \leq y$ then *eventually* the approximate order recognises this; the second half says that if the two elements are *always* approximately ordered such that $x \leq^{n} y$ then they genuinely have that order.

We briefly note here that approximate linear preorders yield uniformly continuous and decidable predicates.

Lemma 4.1.15. [Ƴ] [Ƴ] *Given an approximate linear preorder* $\leq^{-} : \mathbb{N} \to X \to X \to \Omega$ *on a closeness space* X*, the predicates*

$$p_l^{y,\varepsilon}(x) := x \leq^{\varepsilon} y \text{ and } p_r^{y,\varepsilon}(x) := y \leq^{\varepsilon} x$$

are uniformly continuous and decidable for any $y : X$.

Proof. The decidability of the predicates is immediate from Definition 4.1.13.(i). The modulus of uniform continuity for both predicates is $\varepsilon : \mathbb{N}$: we show this for $p_l^{y,\varepsilon}$; the proof for $p_r^{y,\varepsilon}$ is identical.

Given $x_1, x_2 : X$ such that $C_{\varepsilon}(x_1, x_2)$, we want to show that $x_1 \leq^{\varepsilon} y$ implies $x_2 \leq^{\varepsilon} y$. By the closeness of x_1 and x_2, we trivially have $x_2 \leq^{\varepsilon} x_1$ from Definition 4.1.13.(ii). The result then follows by the approximate linear preorder's transitivity.

Returning to our motivating example, we show that sequences of finite linearly ordered types have an approximate linear preorder.

Definition 4.1.16. [Ƴ] Given a preorder $\leq_D : D \to D \to \Omega$ on a discrete set D, the *approximate lexicographic order* $\leq^{-}_{D^{\mathbb{N}}} : \mathbb{N} \to D^{\mathbb{N}} \to D^{\mathbb{N}} \to \Omega$ on $D^{\mathbb{N}}$ is defined:

$$\begin{aligned} \leq^{\varepsilon}_{D^{\mathbb{N}}} &: \mathbb{N} \to D^{\mathbb{N}} \to D^{\mathbb{N}} \to \Omega, \\ \alpha \leq^{\varepsilon}_{D^{\mathbb{N}}} \beta &:= \Pi_{(n:\,\mathbb{N})}\,(n < \varepsilon \to \alpha \sim^{n} \beta \to \alpha_n \leq_D \beta_n)\,. \end{aligned}$$

Lemma 4.1.17. [Ɏ] *Given a linear order $\leq_D : D \to D \to \Omega$ on a discrete set D, the approximate lexicographic order $\leq^{-}_{D^{\mathbb{N}}} : \mathbb{N} \to D^{\mathbb{N}} \to D^{\mathbb{N}} \to \Omega$ is an approximate linear preorder.*

Proof. [f] The proof that $\alpha \leq^{\varepsilon}_{D^{\mathbb{N}}} \beta$ is a preorder is almost identical to Lemma 4.1.10, and so we avoid repeating it here. In order to prove this preorder is linear, we proceed by induction on $\varepsilon : \mathbb{N}$.

In the case where $\varepsilon := 0$ then both $\alpha \leq^{0}_{D^{\mathbb{N}}} \beta$ and $\beta \leq^{0}_{D^{\mathbb{N}}} \zeta$ are trivially proved. We arbitrarily choose the former and show that for all $i : \mathbb{N}$ such that $i < 0$ and $\alpha \sim^{i} \beta$ we have $\alpha_i \leq \beta_i$ vacuously holds because $i < 0$ is empty.

In the case where $\varepsilon := \varepsilon' + 1$, we proceed by the linearity of the linear order on D – i.e. we check which side of $(\alpha_0 \leq_D \beta_0) + (\beta_0 \leq_D \alpha_0)$ and the inductive hypothesis – i.e. we check which side of $\left(\text{tail } \alpha \leq^{\varepsilon'}_{D^{\mathbb{N}}} \text{tail } \beta\right) + \left(\text{tail } \beta \leq^{\varepsilon'}_{D^{\mathbb{N}}} \text{tail } \alpha\right)$ holds.

In the case where $\alpha_0 \leq_D \beta_0$ and tail $\alpha \leq^{\varepsilon'}_{D^{\mathbb{N}}}$ tail β then it is straightforward to show that $\alpha \leq^{\varepsilon}_{D^{\mathbb{N}}} \beta$. The case where $\beta_0 \leq_D \alpha_0$ and tail $\beta \leq^{\varepsilon'}_{D^{\mathbb{N}}}$ tail α is similarly straightforward.

In the case where $\alpha_0 \leq_D \beta_0$ and tail $\beta \leq^{\varepsilon'}_{D^{\mathbb{N}}}$ tail α, we use the discreteness of D to check whether or not $\alpha_0 = \beta_0$ holds. If it does, then also $\beta_0 \leq \alpha_0$ by reflexivity of the linear order on D, and hence $\beta \leq^{\varepsilon}_{D^{\mathbb{N}}} \alpha$. If it does not, then we show that $\alpha \leq^{\varepsilon}_{D^{\mathbb{N}}} \beta$ by induction on the $i : \mathbb{N}$ that is such that $i < \varepsilon$ and $\alpha \sim^{i} \beta$ in order to show that $\alpha_i \leq \beta_i$.

The case where $i := 0$ is trivial as we already have $\alpha_0 \leq_D \beta_0$.

The case where $i := i' + 1$ is vacuous as there is a contradiction between $\alpha \sim^{i'+1} \beta$ and $\alpha_0 \neq \beta_0$.

The case where $\beta_0 \leq_D \alpha_0$ and tail $\alpha \leq^{\varepsilon'}_{D^{\mathbb{N}}}$ tail β uses the same technique as the previous case.

Following this, we prove the numbered conditions of Definition 4.1.13:

(i) In order to prove that the linear preorder is decidable, we again proceed by induction on $\varepsilon : \mathbb{N}$. Recall from the above that the base case is trivial; in the case where $\varepsilon := \varepsilon' + 1$ we use the induction hypothesis to check whether or not $\alpha \leq^{\varepsilon'}_{D^{\mathbb{N}}} \beta$ holds. If it does not, then $\alpha \leq^{\varepsilon}_{D^{\mathbb{N}}} \beta$ clearly does not hold. If it does, we next check whether or not $\alpha \sim^{\varepsilon'} \beta$ holds (by Lemma 2.4.12).

In the case where $\alpha \sim^{\varepsilon'} \beta$ holds, we further check whether or not $\alpha_{\varepsilon'} \leq \beta_{\varepsilon'}$ (by the fact that any linear order on a discrete type is decidable). If it does not, then $\alpha \leq^{\varepsilon}_{D^{\mathbb{N}}} \beta$ clearly does not hold. If it does, then $\alpha \leq^{\varepsilon}_{D^{\mathbb{N}}} \beta$ holds;

i.e. if there is an $i \colon \mathbb{N}$ that is such that $i < \varepsilon$ and $\alpha \sim^i \beta$ then $\alpha_i \leq \beta_i$. If $i < \varepsilon'$ then $\alpha_i \leq \beta_i$ immediately follows by $\alpha \leq^{\varepsilon'}_{D^{\mathbb{N}}} \beta$ and $\alpha \sim^{\varepsilon'} \beta$. If $i = \varepsilon'$, then the result is immediate as we have $\alpha_{\varepsilon'} \leq \beta_{\varepsilon'}$.

In the case where $\alpha \sim^{\varepsilon'} \beta$ does not hold, then $\alpha \leq^{\varepsilon}_{D^{\mathbb{N}}} \beta$ holds; i.e. if there is an $i \colon \mathbb{N}$ that is such that $i < \varepsilon$ and $\alpha \sim^i \beta$ then $\alpha_i \leq \beta_i$. If $i < \varepsilon'$ then then, as above, the result is immediate. If $i = \varepsilon'$, then result is vacuous as there is a contradiction between the assumptions that both $\neg\alpha \sim^{\varepsilon'} \beta$ and $\alpha \sim^i \beta$.

(ii) By Lemma 3.2.59, the assumption that $C_\varepsilon(\alpha, \beta)$ is propositionally equivalent to $\alpha \sim^\varepsilon \beta$. Therefore we assume the latter statement and want to show that if there is an $i \colon \mathbb{N}$ that is such that $i < \varepsilon$ and $\alpha \sim^i \beta$ then $\alpha_i \leq \beta_i$. Beecause $i < \varepsilon$, the assumption tells us that $\alpha_i = \beta_i$, and therefore the result follows by reflexivity of the linear order on D.

Lemma 4.1.18. [$\mathbb{V}$] *Given a discrete set D, the approximate lexicographic order $\leq^{-}_{D^{\mathbb{N}}} \colon \mathbb{N} \to D^{\mathbb{N}} \to D^{\mathbb{N}} \to \Omega$ relates to the lexicographic order $\leq_{D^{\mathbb{N}}} \colon D^{\mathbb{N}} \to D^{\mathbb{N}} \to \Omega$.*

Proof. [t] We prove the numbered conditions of Definition 4.1.14:

(i) Assuming $\alpha \leq_{D^{\mathbb{N}}} \beta$, then for all $i \colon \mathbb{N}$ such that $\alpha \sim^i \beta$ we have $\alpha_i \leq \beta_i$. This means that $\alpha \leq^{\varepsilon}_{D^{\mathbb{N}}} \beta$ for *all* $\varepsilon \colon \mathbb{N}$, and so by setting $n := 0$ we have $\sum_{(n \colon \mathbb{N})} \prod_{(\varepsilon \colon \mathbb{N})} (n < \varepsilon \to x \leq^\varepsilon y)$. The result then follows by truncating this final proof term.

(i) Assuming $\alpha \leq^{n}_{D^{\mathbb{N}}} \beta$ holds for all $n \colon \mathbb{N}$, we need to show that given any $i \colon \mathbb{N}$ such that $\alpha \sim^i \beta$ we have $\alpha_i \leq \beta_i$. This is easy: for the given i we simply use the proof that $\alpha \leq^{i+1}_{D^{\mathbb{N}}} \beta$.

Therefore, there are indeed infinite types that we can perform global optimisation on, so long as we use closeness spaces (or metric spaces, as in the case of the real numbers) to utilise approximate linear preorders. We illuminate this in Section 4.1.2.

Before doing that — for the later purposes of regression in Section 4.2 wherein we will optimise functions with co-domain $\mathbb{N}_\infty$ — we show that the extended naturals have an approximate lexicographic order that coincides with the established order on them (as defined in Definition 3.2.13).

Definition 4.1.19. [$\mathbb{V}$] [$\mathbb{V}$] Given a preorder $\leq \colon X \to X \to \Omega$ and an approximate linear preorder $\leq^{-} \colon \mathbb{N} \to X \to X \to \Omega$ on a closeness space X, and a type family $P \colon X \to \mathcal{V}$, we define the *inclusion order* on the type ΣP and the *inclusion approximate* *order* on the corresponding closeness space (defined in Corollary 3.2.52) by the following:

$$\begin{aligned}(x, px) \leq (y, py) &:= x \leq y,\\ (x, px) \leq^{\varepsilon} (y, py) &:= x \leq^{\varepsilon} y,\end{aligned}$$

Lemma 4.1.20. [ᚠ] [ᚠ] *Given a preorder $\leq : X \to X \to \Omega$ and an approximate linear preorder $\leq^{-} : \mathbb{N} \to X \to X \to \Omega$ on a closeness space X, and a truth-valued type family $P : X \to \Omega$, the inclusion approximate order on the closeness space ΣP is indeed an approximate linear preorder which relates to the inclusion preorder on ΣP.*

Proof (Sketch). [t] Because the definition of the inclusion orders simply compares the first arguments by the original orders — and discards the second arguments — each property we are required to prove for the inclusion orders is immediate from the fact the original orders satisfy those same properties.

Corollary 4.1.21. [ᚠ] *There is an approximate lexicographic order on $\mathbb{N}_\infty$ which relates to the standard lexicographic preorder (as defined in Definition 3.2.13).*

Proof. [t] By Lemmas 4.1.17, 4.1.18 and 4.1.20.

4.1.2 Generalised global optimisation

The definition of a global minimum argument of a function whose domain is linearly ordered is intuitive, and we can always compute a global minimum argument of such a function if the codomain is pointed and finite linearly ordered.

Definition 4.1.22. [ᚠ] For any type X and preorder $\leq : Y \to Y \to \Omega$ on a type Y, an element $x_0 : X$ is a *global minimum argument* of a function $f : X \to Y$ if $f(x_0) \leq f(x)$ for all $x : X$.

Lemma 4.1.23. [ᚠ] *For any pointed finite linearly ordered type X and linear preorder $\leq : Y \to Y \to \Omega$ on a type Y, every function $f : X \to Y$ has a global minimum argument.*

Proof. Recall from Definition 2.4.4 of finite linearly ordered types that $X \simeq \mathsf{Fin}(n)$ for some $n \colon \mathbb{N}$, and therefore that (by Definition 2.3.21) there is some equivalence $g \colon X \to \mathsf{Fin}(n)$.

We proceed by induction on n. When $n := 0$ then $X \simeq \mathbb{0}$, which is not pointed and therefore the result follows vacuously. When $n := 1$ then $X \simeq \mathbb{0} + \mathbb{1}$, and thus X has only one element $g(\mathsf{inr}\ \star) \colon X$ which must, by reflexivity of the linear preorder, be the global minimum argument of f.

When $n := n' + 1$ then $X \simeq \mathsf{Fin}(n') + \mathbb{1}$. We use the inductive hypothesis to find the global minimum argument $x \colon \mathsf{Fin}(n')$ of the function $(f \circ g \circ \mathsf{inl}) : \mathsf{Fin}(n') \to Y$. This means that only $g(\mathsf{inl}\ x) \colon X$ or $g(\mathsf{inr}\ \star) \colon X$ can be the global minimum argument to f. We therefore use the linearity of the linear preorder to find out which side of $(f(g(\mathsf{inl}\ x)) \le f(g(\mathsf{inr}\ \star)) + (f(g(\mathsf{inr}\ \star)) \le f(g(\mathsf{inl}\ x)))$ holds — if the left-hand side holds then $f(g(\mathsf{inl}\ x)) \le f(x)$ for all $x \colon X$, while if the right-hand side holds then $f(g(\mathsf{inr}\ \star)) \le f(x)$ for all $x \colon X$.

Replacing the order with an approximate linear preorder, we now state the general version of the global optimisation problem using closeness spaces. Similarly to the above, it is the case that any function on pointed finite linearly ordered codomain and whose domain has an approximate linear preorder has a computable global minimum up to any degree of precision.

Definition 4.1.24. [🐓] For any type X and approximate linear preorder $\le^- \colon \mathbb{N} \to Y \to Y \to \Omega$ on a closeness space Y, an element $x_0 \colon X$ is an *ε-global minimum argument* of a function $f \colon X \to Y$, given any precision $\varepsilon \colon \mathbb{N}$, if $f(x_0) \le^\varepsilon f(x)$ for all $x \colon X$.

Lemma 4.1.25. [🐓] *For any pointed finite linearly ordered type X and approximate linear preorder $\le^- \colon \mathbb{N} \to Y \to Y \to \Omega$ on a closeness space Y, every function $f \colon X \to Y$ has an ε-global minimum argument given any precision $\varepsilon \colon \mathbb{N}$.*

Proof. By definition of approximate linear preorders (Definition 4.1.13), the relation $\le^\epsilon \colon X \to X \to \Omega$ is a linear preorder. Therefore, the result immediately follows by Lemma 4.1.23.

As we saw in Definition 4.1.3, real global optimisation is performed on *compact* intervals of the real numbers. Recall that a metric space is compact if it is totally bounded and complete, and that there is a relationship between compact spaces and searchable sets. As we are only minimising uniformly continuous functions, we do not require completeness[3]. Furthermore, uniformly continuous searchability (see Definition 3.3.3) is inappropriate for global optimisation. This is because, in order to prove the search condition when searching for a global minimum argument, we would have to prove that such an argument exists anyway. Therefore, our generalised global optimisation theorem takes place on totally bounded closeness spaces.

By using the totally bounded (see Definition 3.2.35) and uniformly continuous (see Definition 3.2.27) properties, we reduce the problem of searching the potentially infinite space X for an ε-global minimum into that of searching a finite ε-net (see Definition 3.2.34) for an element which represents a ε-global minimum. This approach is intuitive, and reflects approaches to arbitrary-precision global optimisation in the literature [FG09]. Furthermore, it is exactly the same process as what was performed in Section 3.3, wherein we used uniform continuity to search a potentially-infinite space.

Theorem 4.1.26. [ᚠ] *Given a pointed totally bounded closeness space X and approximate linear preorder $\leq^{-} : \mathbb{N} \to Y \to Y \to \Omega$ on a closeness space Y, any uniformly continuous function $f : X \to Y$ has an ε-global minimum given any precision $\varepsilon : \mathbb{N}$.*

Proof. Given the requested precision $\varepsilon : \mathbb{N}$, by total boundedness we obtain a δ-net X' of X, where $\delta : \mathbb{N}$ is the modulus of uniform continuity of f for ε. Recall from Definition 3.2.34 that X' is such that there are $g : X' \to X$ and $h : X \to X$ such that for all $x : X$ we have $C_\delta(x, g(h(x)))$.

By Lemma 4.1.25, we can compute an ε-global minimum of the function $f \circ g : X' \to Y$, i.e. we have $x'_0 : X'$ such that $f(g(x'_0)) \leq^{\varepsilon} f(g(x'))$ for all $x' : X'$.

Then, given any $x : X$, it is the case that $C_\delta(x, g(h(x))$ and thus, by Definition 4.1.13.(i), that $f(g(h(x))) \leq^{\varepsilon} f(x)$.

Therefore, given any $x : X$, we have $f(g(x'_0)) \leq^{\varepsilon} f(g(h(x))) \leq^{\varepsilon} f(x)$ and thus, by transitivity of the approximate linear preorder, $g(x'_0) : X$ is an ε-global minimum argument of f.

Finally, we apply our theorem to our motivating example, and show that we can optimise functions on sequences of finite linearly ordered types via their lexicographic orders.

[3]Informally, we do not require completeness due to the fact that uniformly continuous functions on metric spaces extend uniquely to the completion of that space and, further, the completion of a totally bounded metric space is compact [TD88].

Corollary 4.1.27. *Given finite linearly ordered types F and G, with F pointed, any uniformly continuous function $f\colon F^{\mathbb{N}} \to G^{\mathbb{N}}$ has, for any requested precision $\varepsilon\colon \mathbb{N}$, an ε-global minimum via discrete-sequence closeness spaces (see Definition 3.2.56) and the approximate lexicographic ordering on $G^{\mathbb{N}}$.*

Proof. Recall that, by Corollary 3.2.60, the discrete-sequence closeness space of a finite linearly ordered type F is totally bounded. The result then follows from Theorem 4.1.26 by Lemma 4.1.17.

4.2 Parametric Regression

The work of this section was previously published as part of a joint paper with Dan R. Ghica at the *Logic in Computer Science (LICS) 2021* conference [GA21].

Parametric regression analysis is a set of algorithms for estimating the *relationship* between a dependent variable y (outcome) and several independent variables $\{x_0, \ldots, x_{n-1}\}$ (predictors), where the outcome is a function of the observations that has potentially, and indeterminably, been *distorted*. A parameterised function is proposed as a *model* for this function. The value of the parameters is then computed such that, given finitely-many predictor-outcome observations, a *loss function* between the observed outcomes and those estimated by the model on input of the predictors is minimised [YS09].

Therefore, parametric regression is the problem of finding — given finitely-many predictor-outcome observations — (approximations of) best choice parameters for a given parameterised model function using a given loss function. As with global optimisation in Section 4.1, we first state this problem using real numbers.

Definition 4.2.1. [🕮] Given $n, i, d\colon \mathbb{N}$, some loss function $L\colon \mathbb{R} \to \mathbb{R} \to \mathbb{R}_{\geq 0}$, some parameterised model function $M\colon \mathbb{R}^i \to (\mathbb{R}^d \to \mathbb{R})$ and finitely-many predictor-outcome observations $\{(xs_0, y_0), \ldots, (xs_{n-1}, y_{n-1})\}\colon (\mathbb{R}^d \times \mathbb{R})^n$, the parameters $ps^*\colon \mathbb{R}^i$ are *best choice* if they minimise the loss between the outcomes estimated by the regressed function $M_{ps}\colon \mathbb{R}^d \to \mathbb{R}$ and the observed outcomes; i.e. if they are a global minimum of the function $\left(\lambda(ps\colon \mathbb{R}^i).\ \sum_{j=0}^{n} L(M_{ps}(xs_i), y_i)\right) : \mathbb{R}^i \to \mathbb{R}_{\geq 0}$.

The choice of a particular loss function — i.e. a pseudometric on the function space $(\mathbb{R}^d \to \mathbb{R})$ — is a field in its own right, but a common example is the least-squares method $L(x, y) := d_{\mathbb{R}}(x, y)^2$. As an example of a particular model function, consider *linear regression* where the model $M\colon \mathbb{R}^2 \to (\mathbb{R} \to \mathbb{R})$ is defined $M_{(\alpha,\beta)}(x) := \alpha x + \beta$.

Based on the above rationalisation of the parametric regression problem, we view it as an instance of the global optimisation problem explored in the previous section. As with global optimisation, therefore, we cannot in general compute a best choice set of parameters — but we can compute parameters that are, for some degree of precision, *approximately best choice.*

Definition 4.2.2. [🕮] Given $n, i, d \colon \mathbb{N}$, some loss function $L \colon \mathbb{R} \longrightarrow \mathbb{R} \longrightarrow \mathbb{R}_{\geq 0}$, some parameterised model function $M \colon \mathbb{R}^i \longrightarrow (\mathbb{R}^d \longrightarrow \mathbb{R})$, finitely-many predictor-outcome observations $\{(xs_0, y_0), ..., (xs_{n-1}, y_{n-1})\} \colon (\mathbb{R}^d \times \mathbb{R})^n$ and any precision $\varepsilon \colon \mathbb{R}_{\geq 0}$, the parameters $ps^* \colon \mathbb{R}^i$ are *ε-best choice* if they minimise the loss between the outcomes estimated by the regressed function $M_{ps} \colon \mathbb{R}^d \longrightarrow \mathbb{R}$ and the observed outcomes up-to-ε; i.e. if they are an ε-global minimum of the function $\left(\lambda(ps \colon \mathbb{R}^i).\ \sum_{j=0}^{n} L(M_{ps}(xs_i), y_i)\right) \colon \mathbb{R}^i \longrightarrow \mathbb{R}_{\geq 0}$.

4.2.1 Generalised parametric regression

By generalising Definition 4.2.2, we eliminate the need to specify the arity of our parameter/predictor spaces. We replace the notion of an arbitrary loss function such as least-squares by the least-closeness pseudocloseness function (defined in Definition 3.2.69); this means that rather than minimising loss we seek to maximise the least-closeness. Furthermore, we replace the idea of predictor-outcome observations $\{(xs_0, y_0), ..., (xs_{n-1}, y_{n-1})\} \colon (X \times Y)^n$ by a combination of predictor outcomes $\{x_0, ..., x_{n-1}\} \colon X^n$ and a function $\mathcal{O} \colon X \longrightarrow Y$ which can be thought of as the *oracle* of the outcome observations — the (potentially distorted) function from which they arise on input of the predictors; i.e. $\mathcal{O}(xs_i) = y_i$.

Definition 4.2.3. [ᚠ] Given a function $f \colon X \longrightarrow Y$ where Y is a pre-ordered type, $x_0 \colon X$ is a *global maximum* of f if for all $x \colon X$ we have $f(x_0) \geq f(x)$.

Definition 4.2.4. Given a type of predictors X, closeness space of outcomes Y, type of parameters P, some parameterised model function $M \colon P \longrightarrow (X \longrightarrow Y)$, finitely-many predictor observations $\{x_0, ..., x_{n-1}\} \colon X^n$ (for some $n \colon \mathbb{N}$), some oracle function $\mathcal{O} \colon X \longrightarrow Y$ and any precision $\varepsilon \colon \mathbb{N}$, the parameter $p^* \colon P$ is *ε-best choice* if it maximises the least-closeness pseudocloseness between the predictors' outcomes as estimated by the regressed function $M_{p^*} \colon X \longrightarrow Y$ and as observed from the oracle up-to-ε; i.e. if p^* is an ε-global maximum of the function $\left(\lambda(p \colon P).\mathsf{min}(c_Y(M_p(x_0), \mathcal{O}(x_0)), ..., c_Y(M_p(x_{n-1}), \mathcal{O}(x_{n-1}))\right) \colon P \longrightarrow \mathbb{N}_\infty$.

In this way, we have now re-imagined parametric regression as the process of approximating a black-box *oracle* with a *parameterised model* using a *pseudocloseness function*. Therefore, our generalised parametric regression not only (i) generalises from reals to closeness spaces, but also (ii) *methodologically* generalises the problem of regression.

We can go one step further and remove explicit observations of the oracle altogether, allowing us to generalise the type of oracles themselves from function spaces to pseudocloseness spaces. By doing this, instead of using the least-closeness pseudocloseness function, we use the pseudocloseness function that the pseudocloseness space of oracles is equipped with.

Definition 4.2.5. Given a type of parameters P, a pseudocloseness space of oracles O, some oracle $\mathcal{O} : O$ and some parameterised model function $M : P \to O$, the parameter $p^* : P$ is *ε-best choice* if it maximises the closeness between the regressed function M_{p^*} and the oracle $\mathcal{O}$ up-to-ε; i.e. if p^* is an ε-global maximum of the function $\big(\lambda(p : P).c'_O(M_p, \mathcal{O})\big) : P \to \mathbb{N}_\infty$.

We use this second generalisation of regression when stating and proving our convergence theorems for regression in the next subsection. However, for the practical purposes of Chapter 6, we return to the less general Definition 4.2.4, which defines regression on function spaces by using the least-closeness pseudocloseness function. We note here that, as Definition 4.2.5 is a generalisation of Definition 4.2.4, the convergence theorems proved on the former still hold for the latter.

4.2.2 Convergence theorems for parametric regression

Convergence properties of interpolation, another way of constructing models out of data, have been studied extensively by Weierstrass-style theorems [Pin00]. In this section, we make a methodological contribution by providing several convergence properties of our generalised variant of parametric regression (Definition 4.2.5) using optimisation and search. The contribution here is more methodological than technical, as the proofs follow naturally from the structures and theorems of Chapters 3 and 4. The statements on the other hand are not obvious and require a different conceptual, not just mathematical, perspective on parametric regression analysis.

Convergent parametric regression via global optimisation

Guarantees that can be made on computing an ε-global minimum of the loss function can be automatically considered as guarantees on the precision of the regressed model.

Recall that the convergence of interpolation guarantees that any oracle (subject to hygiene conditions) can be reconstituted to any desired precision if the number of samples is large enough [Pin00]. A similar theorem cannot hold for regression, for the simple reason that in regression we must commit to a model which may or may not be similar to the oracle function. For example, if our oracle has quadratic behaviour, no amount of data will yield a precise linear approximation of it. This commitment to a particular model must be taken into account in formulating convergence properties for regression. If the model is completely wrong then, of course, convergence cannot be achieved.

However, even in the case where the model is incorrect, we can still employ generalised global optimisation (as long as our types permit that) in order to compute an ε-best choice parameter of the model up to any precision.

Theorem 4.2.6 (Regression as minimisation)**.** [ᚠ] *Given a totally bounded closeness space of parameters P, pseudocloseness space of oracles O oracle $\mathcal{O} : O$, and any parameterised uniformly continuous model function $M : P \to O$, we can compute an ε-best choice parameter for M given every precision $\varepsilon : \mathbb{N}$.*

Proof. Recall that $\mathbb{N}_\infty$ is a closeness space (Corollary 3.2.53) and has an approximate lexicographic order (Corollary 4.1.21). Therefore, we can perform ε-global maximisation[a] on it using Theorem 4.1.26. We then compute an ε-global maximum of the function $\left(c'_O(\mathcal{O}, M(p))\right) : P \to \mathbb{N}_\infty$, where $c'_O : O \to O \to \mathbb{N}_\infty$ is the pseudocloseness function on O. The function that we maximise is uniformly continuous by the ultrametric property of the pseudocloseness space (Definition 3.2.68).

[a]Maximisation can be achieved by the same theorem as minimisation by simply swapping the order in which elements are evaluated by the approximate order.

Convergent regression via uniformly continuous search

Our first convergence theorem stated that we can compute a ε-best choice parameter given minimal conditions on the oracle and parameter types, as well as the model function. We now consider computing parameters for regression that are not necessarily ε-best choice, but which maximise the pseudocloseness to an acceptable level parameterised by ε. These convergence theorems require additional conditions, but are more 'practical' in the sense that we do not necessarily have to exhaust all possible candidate solutions in order to return an acceptable parameter for the given ε (as we see later in the examples of Section 6.1.3, such as Example 6.1.35).

A general and absolute guarantee of precision can only be given if the model is the same as the oracle up to the value of the model parameters; i.e. the model is chosen correctly, and the oracle is not distorted. If this is the case then, by using uniformly continuous search, we can indeed maximise the pseudocloseness between the regressed and true oracles and, further, we can make it arbitrarily large.

To express this property we introduce the concept of a *synthetic oracle*, which is simply an oracle $\mathcal{O}$ "synthesised" from a model function M by applying it to an arbitrary and unknown parameter k; i.e. $\mathcal{O} := M(k)$.

Definition 4.2.7. Given type of parameters P and type of oracles O, an oracle $\mathcal{O} : O$ is *synthetically constructed from* M if there is some parameter choice $p : P$ such that $\mathcal{O} = M_p$.

Definition 4.2.8. [⚐] Given a uniformly continuously searchable type of parameters P and pseudocloseness space of oracles O, we define the *parametric regressor* function $\mathrm{reg} : \mathbb{N} \to (P \to O) \to O \to P$ as,

$$\mathrm{reg}(\varepsilon, M, \mathcal{O}) := \mathcal{E}_P \left(\lambda(p : P).C_\varepsilon(M_p, \mathcal{O})\right),$$

where $\mathcal{E}_P : \text{decidable-uc-predicate}(P) \to P$ is the uniformly continuous searcher on P (as introduced in Definition 3.3.2).

Theorem 4.2.9 (Convergence of distortion-free regression). [⚐] *Given a uniformly continuously searchable type of parameters P, pseudocloseness space of oracles O, parameterised uniformly continuous model function $M : P \to O$ and oracle $\mathcal{O} : O$ synthetically constructed from M, we can use the parametric regressor* $\mathrm{reg} : \mathbb{N} \to (P \to O) \to O \to P$ *to build the regressed oracle $\omega : O$ using only ε, M and $\mathcal{O}$ (i.e. $\omega := \mathrm{reg}(\varepsilon, M, \mathcal{O})$) such that $C_\varepsilon(\omega, \mathcal{O})$, for any precision $\varepsilon : \mathbb{N}$.*

Proof. The result follows from the later Theorem 4.2.10 by setting $\Psi := \mathrm{id}$; i.e. the distortion function is just the identity function, as the oracle we query is not distorted from the true oracle.

Note that a raw intuition of this theorem statement can be misleading: the regressor sees the synthetic oracle as a black box process, so it is not simply searching for the parameter of the synthetic oracle k. It is searching for *any* parameter that makes the pseudocloseness between the true and regressed oracles ε-large.

The more traditional case is when the oracle *is* prone to some distortion. We model this case in our framework by using a *distortion function* $\Psi : O \to O$ which is applied to the *true synthetic oracle* $\mathcal{O}: O$ to yield the *distorted oracle* $\mathcal{O}_\Psi : O$. It is this distorted oracle that the parametric regressor receives and can query for observations.

We cannot anymore expect to be able to maximise the pseudocloseness between the regressed and true oracles to any degree of precision. However, we *can* guarantee that the pseudocloseness between the regressed and true oracles is bounded by that between the regressed and distorted oracles.

Theorem 4.2.10 (Convergence of distortion-prone regression)**.** [✓] *Given a uniformly continuously searchable type of parameters P, pseudocloseness space of oracles O, parameterised uniformly continuous model function $M: P \to O$, oracle $\mathcal{O}: O$ synthetically constructed from M and distortion function $\Psi: O \to O$, we can use the parametric regressor* $\mathsf{reg}: \mathbb{N} \to (P \to O) \to O \to P$ *to build the regressed oracle $\omega: O$ using only ε, M and $\Psi\mathcal{O}$ (i.e. $\omega := \mathsf{reg}(\varepsilon, M, \Psi\mathcal{O})$) such that if $C_\varepsilon(\Psi\mathcal{O}, \mathcal{O})$ then $C_\varepsilon(\omega, \mathcal{O})$, for any precision $\varepsilon: \mathbb{N}$.*

Proof. We just need to show that $C_\varepsilon(\Psi\mathcal{O}, \omega)$ as then the result will follow by transitivity of the closeness relation (Lemma 3.2.23) and the assumption that $C_\varepsilon(\Psi\mathcal{O}, \mathcal{O})$.

Because $\omega := \mathsf{reg}(\varepsilon, M, \Psi\mathcal{O}) := \mathcal{E}_P(\lambda(p: P).C_\varepsilon(M_p, \mathcal{O}))$, the result follows if there is some $p': P$ such that $C_\varepsilon(M_{p'}, \mathcal{O})$. Because the model is synthetically constructed, we can set p' as the parameter from which it was constructed, and the result follows immediately.

Chapter 5

Real Numbers

We introduced our type-theoretic framework using Agda in Chapter 2, the core concepts of searchability and continuity in Chapter 3, and our generalised variants of global optimisation and parametric regression — algorithms usually defined explicitly on real numbers — in Chapter 4. We now wish to take our work full circle: to program within our framework instantiations of these processes that operate on representations of (compact intervals of) the real numbers.

Constructive approaches to representing the real numbers are well-studied: the Cauchy reals and Dedekind reals are representations that have been previously defined and used in dependent type theory for analysis [Uni13; Boo20]. In practice, however, the Dedekind reals are inconvenient for computation, owing to the fact every real number is represented uniquely by exactly one Dedekind real [GNSW07]. The Cauchy reals do not have such a uniqueness property, and as such have been found to be more convenient for computation: different variations of Cauchy (i.e., convergent) sequences of rational numbers have been used as representations of real numbers for the purpose of performing exact real computation. In this chapter, we introduce two such convenient representations of (Cauchy) real numbers from the literature: ternary signed-digit encodings and ternary Boehm encodings [Di 93; Boe20].

We formalise the structure and some of the algorithms of the signed-digit encodings of the compact interval $[-1, 1]$ within our Agda framework and — going beyond this — prove the correctness of these definitions. In order to achieve this latter goal of verification, we further formalise the Escardó-Simpson interval object, an axiomatic specification of the real numbers which supports constructive mathematics by design [ES01].

In classical mathematics, the real numbers are axiomatised as the unique complete Archimedean field (see e.g. [Tre13]). This approach does not work in constructive mathematics – for example, as we discussed in Section 4.1, the axiom that such reals have a linear order is the analytic LLPO [Shu18]. The alternative use of the interval object in this chapter is therefore appropriate; furthermore, it has the advantage of having fewer operations and axioms and so is easier to work with in practice.

For the Boehm encodings, we formalise and prove the correctness of Boehm's definition; though the correctness of his arithmetic operations is relegated to further work (see Section 7.2.2). For the sake of variation, we use a different approach to verifying the structure of the Boehm encodings, and instead use the Dedekind reals [Bau08]. We seek to show that every ternary Boehm encoding of a real number gives a Dedekind real encoding of that real number.

5.1 Escardó-Simpson interval object

In 2001, Martín Escardó and Alex Simpson proposed a categorical specification of closed real intervals which supports constructive mathematics by design [ES01]. The basic structure is that of bipointed *midpoint algebras*, on which we give a universal property that is a variation of the completeness axiom, which serves as a computation principle for these real numbers.

In their paper, Escardó and Simpson worked in the generality of a category with finite products, but wrote that their specification "applies to a variety of computational settings ... such as intuitionistic type theory" [ES01]. In this section, we contribute to this line of work by formalising the work in constructive type theory using Agda. We will use this type in Section 5.2 in order to verify the signed-digit encodings.

This work was previously published as part of a joint paper with Dan R. Ghica at the *Logic in Computer Science (LICS) 2021* conference [GA21], and given as a talk at the *Workshop on Homotopy Type Theory/Univalent Foundations* (HoTT/UF) [Amb20b; Amb20a].

5.1.1 Cancellative midpoint algebras

We start off by defining the type of midpoint algebras.

Definition 5.1.1. [🕮] A *magma* is a set A equipped with a binary function to itself,

$$\mathsf{Magma}_{\mathcal{U}} := \sum_{(A\colon \mathcal{U})} (\mathsf{is\text{-}set}(A) \times (A \to A \to A)) .$$

For a given magma, we write $(A, \oplus)\colon$ Magma with the proof terms implicit.

Definition 5.1.2. [ ] A magma $(A, \oplus)$ is a *midpoint algebra* if it is,

(i) idempotent, $\prod_{(a\colon A)} (a \oplus a = a)$,

(ii) commutative, $\prod_{(a,b\colon A)} (a \oplus b = b \oplus a)$,

(iii) transpositional, $\prod_{(a,b,c,d\colon A)} ((a \oplus b) \oplus (c \oplus d) = (a \oplus c) \oplus (b \oplus d))$.

For such a structure, we write $(A, \oplus)\colon$ Midpoint-algebra with proof terms (i)-(iii) implicit.

Functions between midpoint algebras that preserve the structure are called midpoint homomorphisms.

Definition 5.1.3. [ ] Given two midpoint algebras $(A, \oplus_A)$ and $(B, \oplus_B)$, a function $h\colon A \to B$ is a *midpoint homomorphism* if it preserves the midpoint operation:

$$\mathsf{is\text{-}midpoint\text{-}hom}((A, \oplus_A), (B, \oplus_B), h) := \prod_{(a,b\colon A)} (h(a \oplus_A b) = h(a) \oplus_B h(b)) .$$

If the midpoint algebras are the same, we write $\mathsf{is\text{-}midpoint\text{-}hom}((A, \oplus_A), h)$ as shorthand.

Lemma 5.1.4. [ ] [ ] *The identity function is a midpoint homomorphism and composition preserves homomorphisms.*

Proof. For the identity function on a midpoint algebra $(A, \oplus_A)$, we need to show that for all $a\colon A$ we have $\mathsf{id}_A(a \oplus_A a) = \mathsf{id}_A(a) \oplus_A \mathsf{id}_A(a)$; this is clearly the case by reflexivity. For composition of functions $f\colon A \to B$ and $g\colon B \to C$ between midpoint algebras $(A, \oplus_A)$, $(B, \oplus_B)$ and $(C, \oplus_C)$, we want to show that $g(f(a_1 \oplus_A a_2)) = g(f(a_1)) \oplus_C g(f(a_2))$ for all $a_1, a_2\colon A$. This is the case because $g(f(a_1 \oplus_A a_2)) = g(f(a_1) \oplus_B f(a_2))$ (because f is a midpoint homomorphism) and then $g(f(a_1) \oplus_B f(a_2)) = g(f(a_1)) \oplus_C g(f(a_2))$ (because g is a midpoint homomorphism).

The midpoint algebras we utilise for the interval object satisfy an additional property called cancellation.

Definition 5.1.5. [ ] A magma $(A, \oplus)$ is *cancellative* if for all $a, b, c \colon A$ it is the case that $a \oplus c = b \oplus c$ implies $a = b$.

$\mathbb{R}^n$ is a cancellative midpoint algebra closed under the binary midpoint function $\lambda(x, y \colon \mathbb{R}^n).\frac{1}{2}(x + y)$, as are various subsets of $\mathbb{R}$, such as the rationals.

5.1.2 Iteration property

Starting from 0 and 1, the midpoint function can be used to generate every dyadic rational point in $[-1, 1]$. In order to generate all rational and irrational numbers, the interval object requires its own version of the classical *completeness* axiom; recall that this informally states that the real line has no "gaps" or "missing points" [Tre13]. This property, which is called *iteration*, states that there is an operator $M \colon A^{\mathbb{N}} \to A$ that gives the 'infinitely iterated' midpoint of a stream of points of A. Formally, this operator is defined by two sub-properties.

Definition 5.1.6. [ ] Given a magma $(A, \oplus)$ and function $M \colon A^{\mathbb{N}} \to A$, the *first iteration sub-property* is defined by:

$$\mathsf{iterative}_1(A, \oplus, M) := \prod_{(\alpha \colon A^{\mathbb{N}})} \left(M(\alpha) = \alpha_0 \oplus M(\mathsf{tail}\ \alpha)\right).$$

Definition 5.1.7. [ ] Given a magma $(A, \oplus)$ and function $M \colon A^{\mathbb{N}} \to A$, the *second iteration sub-property* is defined by:

$$\mathsf{iterative}_2(A, \oplus, M) := \prod_{(\alpha, \beta \colon A^{\mathbb{N}})} \left(\prod_{(i \colon \mathbb{N})} (\beta_i = \alpha_i \oplus \beta_{i+1}) \right) \to \beta_0 = M(\alpha).$$

Definition 5.1.8. [ ] A magma $(A, \oplus)$ is *iterative* if there is a function $M \colon A^{\mathbb{N}} \to A$ such that both iteration sub-properties are satisfied:

$$\mathsf{iterative}(A, \oplus) := \sum_{(M \colon A^{\mathbb{N}} \to A)} \left(\mathsf{iterative}_1(A, \oplus, M) \times \mathsf{iterative}_2(A, \oplus, M)\right)$$

The first sub-property characterises the iteration operator, while the second gives a computation rule for it with respect to a second stream which corresponds to the iteration on the first. Both of these sub-properties are indeed *properties* rather than additional *structure* on the magma; i.e. given any magma $(A, \oplus)$ and function $M \colon A^{\mathbb{N}} \to A$, the types $\mathsf{iterative}_1(A, \oplus, M)$ and $\mathsf{iterative}_2(A, \oplus, M)$ are propositions (Definition 2.3.5), by the fact that A is a set. This further (by Lemma 2.3.11) means that the composition of these two properties is a property for any given magma and function. We now prove that the type $\mathsf{iterative}(A, \oplus)$ itself is a property for any magma $(A, \oplus)$; the proof comes from the fact that for any given magma, any function satisfying both iteration sub-properties is unique.

Lemma 5.1.9. [ᚠ] *Given a magma $(A, \oplus)$ any two functions that satisfy both iteration sub-properties are pointwise-equal.*

Proof. Given two functions $M_1, M_2 \colon \mathbb{I}^{\mathbb{N}} \to \mathbb{I}$ that satisfy the two iteration sub-properties, we want to show that for any $\alpha \colon \mathbb{I}^{\mathbb{N}}$, it is the case that $M_1(\alpha) = M_2(\alpha)$.

To do this, we define a sequence $\beta \colon \mathbb{I}^{\mathbb{N}}$ which gives the behaviour of $M_1(\alpha)$ as a sequence: i.e. $\beta_i := M_1(\lambda n.\alpha_{n+i})$. By the first iteration sub-property on M_1, for any $i \colon \mathbb{N}$ we have that $\beta_i = \alpha_i \oplus \beta_{i+1}$. Therefore by the second iteration sub-property on M_2, we have that $\beta_0 = M_2(\alpha)$; i.e. $M_1(\alpha) = M_2(\alpha)$.

Corollary 5.1.10. [ᚠ] *Given a magma $(A, \oplus)$ the type $\mathsf{iterative}(M, \oplus)$ is a proposition.*

Proof. [f] By Lemma 5.1.9 and function extensionality, any two functions that satisfy both iteration sub-properties are equal. Hence, because both sub-properties are subsingletons, the result follows by Lemma 2.3.11.

From these sub-properties, Escardó and Simpson prove some expected properties about our iterative midpoint operator.

Lemma 5.1.11. [ᚠ] *Given an idempotent magma $(A, \oplus)$ and $M \colon A^{\mathbb{N}} \to A$ which satisfies the second iteration sub-property, M is itself idempotent,*

$$\prod_{(a \colon A)} \left(M(\lambda(- \colon \mathbb{N}).a) = a \right).$$

Proof. By the second iteration sub-property (Definition 5.1.7), if we set $\alpha, \beta := \lambda - .a$ then once we to prove the antecedent — that for all $i \colon \mathbb{N}$ we have $(\lambda - .a)_i = (\lambda - .a)_i \oplus (\lambda - .a)_i$ — the result will follow. The antecedent just asks us to show that $a = a \oplus a$ holds, which is by the idempotency of $\oplus$ (Definition 5.1.2).

Lemma 5.1.12. [ᚠ] *Given a transpositional magma $(A, \oplus)$ that is iterative by $M \colon A^{\mathbb{N}} \to A$, it is the case that M satisfies the following homomorphic property,*

$$\prod_{(\theta,\zeta \,:\, A^{\mathbb{N}})} (M(\theta) \oplus M(\zeta) = M(\lambda(i \colon \mathbb{N}).\theta_i \oplus \zeta_i))\,.$$

Proof (Sketch). By the second iteration sub-property (Definition 5.1.7), if we set $\alpha := \lambda i.\theta_i \oplus \zeta_i$ and $\beta := \lambda i.M(\lambda n.\theta_{n+i}) \oplus M(\lambda n.\zeta_{n+i})$ then once we prove the antecedent — that for all $i \colon \mathbb{N}$ we have $M(\lambda n.\theta_{n+i}) \oplus M(\lambda n.\zeta_{n+i}) = (\theta_i \oplus \zeta_i) \oplus (M(\lambda n.\theta_{n+i+1}) \oplus M(\lambda n.\zeta_{n+i+1}))$ — the result will follow. The antecedent follows by the first iteration sub-property (Definition 5.1.6) and the transpositionality of $\oplus$ (Definition 5.1.2).

Lemma 5.1.13. [ᚠ] *Given a transpositional magma $(A, \oplus)$ that is iterative by $M \colon A^{\mathbb{N}} \to A$, it is the case that M is symmetric,*

$$\prod_{(\theta \colon (A^{\mathbb{N}})^{\mathbb{N}})} \big(M(\lambda i.M(\lambda j.\theta_{i,j})) = M(\lambda i.M(\lambda j.\theta_{j,i}))\big)\,.$$

Proof (Sketch). [f] By the second iteration sub-property (Definition 5.1.7), if we set $\alpha := \lambda n.M(\lambda j.\theta_{j,n})$ and $\beta := \lambda n.M(\lambda i.M(\lambda j.\theta_{i,j+n}))$ then once we prove the antecedent — that for all $n \colon \mathbb{N}$ we have $M(\lambda i.M(\lambda j.\theta_{i,j+n})) = M(\lambda j.\theta_{j,n}) \oplus M(\lambda i.M(\lambda j.\theta_{i,j+n+1}))$ — the result will follow. The antecedent follows by the first iteration sub-property (Definition 5.1.6) and the above homomorphic property (Lemma 5.1.12).

For the specific details of the latter two proofs, we invite the interested reader to view the formalisation.

The iteration property also gives us the notion of an iterated midpoint homomorphism (or 'M-homomorphism'). Any midpoint homomorphism is automatically an M-homomorphism.

Definition 5.1.14. [ᚠ] Given midpoint algebras $(A, \oplus_A)$ and $(B, \oplus_B)$ that are iterative by functions $M_A \colon A^{\mathbb{N}} \to A$ and $M_B \colon B^{\mathbb{N}} \to B$, a function $h \colon A \to B$ is an *iterated midpoint homomorphism* if,

$$\text{is-M-hom}((A, \oplus_A), (B, \oplus_B), M_A, M_B, h) := \prod_{(\alpha \colon A^{\mathbb{N}})} (h(M_A(\alpha)) = M_B(\lambda n.h(\alpha_n)))\,.$$

If the midpoint algebras are the same, we write $\text{is-M-hom}((A, \oplus_A), M, f)$ as shorthand.

Lemma 5.1.15. [⚐] *Given midpoint algebras* $(A, \oplus), (B, \oplus)$: Midpoint-algebra *that are iterative by functions* $M_A \colon A^{\mathbb{N}} \to A$ *and* $M_B \colon B^{\mathbb{N}} \to B$, *if a function* $h \colon A \to B$ *is a midpoint homomorphism then it is also an iterated midpoint homomorphism.*

Proof. In order to show that $h(M_A(\alpha)) = M_B(\lambda n.h(\alpha_n))$ for any $\alpha \colon A^{\mathbb{N}}$ we first define a sequence $\beta \colon B^{\mathbb{N}}$ which gives the behaviour of $h(M_A(\alpha)) \colon B$ as a sequence: i.e. $\beta_i := h(M_A(\lambda n.\alpha_{n+1}))$. We next show for all $i \colon \mathbb{N}$ we have $\beta_i = h(\alpha_i) \oplus \beta_{i+1}$. Once, we have done this, our result will follow by the second iteration sub-property (Definition 5.1.7).

$$\begin{aligned}
& & \beta_i & \\
:= & & h(M_A(\lambda n.\alpha_{n+i})) & \\
= & & h(\alpha_i \oplus M_A(\lambda n.\alpha_{n+i+1})) & \quad \text{by the first iteration sub-property ((Definition 5.1.6),} \\
= & & h(\alpha_i) \oplus h(M_A(\lambda n.\alpha_{n+i+1})) & \quad \text{by the fact } h \text{ is an } \oplus\text{-homomorphism,} \\
:= & & h(\alpha_i) \oplus \beta_{i+1} & \quad \text{by (i).}
\end{aligned}$$

5.1.3 Finite approximations

We formalise one final structure for iterative midpoint algebras: *finite approximations*, the existence of which is equivalent to having the cancellation property (Definition 5.1.5) [ES01].

Definition 5.1.16. [⚐] Given a midpoint algebra $(A, \oplus)$, two sequences $\alpha, \beta \colon A^{\mathbb{N}}$ are *n-approximately equal*, for a given $n \colon \mathbb{N}$, if,

$$\sum_{(w,z \colon A)} (\alpha_0 \oplus (\alpha_1 \oplus ...(\alpha_{n-1} \oplus w)) = \beta_0 \oplus (\beta_1 \oplus ...(\beta_{n-1} \oplus z)))\,.$$

Definition 5.1.17. [⚐] A midpoint algebra $(A, \oplus)$ that is iterative by $M \colon A^{\mathbb{N}} \to A$ has *finite approximations* if given two sequences $\alpha, \beta \colon A^{\mathbb{N}}$ that are n-approximately equal for all $n \colon \mathbb{N}$, then $M(\alpha) = M(\beta)$.

We formalise the more straightforward direction first: that having finite approximations implies the cancellation property.

Lemma 5.1.18. [⚐] *A midpoint algebra* $(A, \oplus)$: Midpoint-algebra *that is iterative by* $M \colon A^{\mathbb{N}} \to A$ *and has finite approximations is cancellative.*

Proof. Given $a, b, c \colon A$ such that $(a \oplus c) = (b \oplus c)$, we wish to show that $a = b$. This follows from the idempotency of M (Lemma 5.1.11) once we show that $M(\lambda - .a) = M(\lambda - .b)$, which we will show using the finite approximation property $Definition$ 5.1.16. Therefore, we simply need to show that for any $n \colon \mathbb{N}$, we have $(\lambda - .a)_0 \oplus ((\lambda - .a)_1 \oplus ...((\lambda - .a)_{n-1} \oplus c)) = (\lambda - .b)_0 \oplus ((\lambda - .b)_1 \oplus ...((\lambda - .b)_{n-1} \oplus c))$. We proceed by induction on n. The first base case where $n := 0$ is trivial, as we only need to show that $c = c$; the second base case where $n := 1$ requires us to show that $(a \oplus c) = (b \oplus c)$, which we have already assumed.

The inductive case where $n := n' + 1$ for some $n' \colon \mathbb{N}$ requiers us to show that $a \oplus (a \oplus wa) = b \oplus (b \oplus wb)$ where $wa := (\lambda - .a)_0 \oplus ((\lambda - .a)_1 \oplus ...((\lambda - .a)_{n'-1} \oplus c))$ and $wb := (\lambda - .b)_0 \oplus ((\lambda - .b)_1 \oplus ...((\lambda - .b)_{n'-1} \oplus c))$. By the inductive hypothesis, both $a \oplus wa = b \oplus wb$ and $wa = wb$; the result then follows by the below equational reasoning:

$$\begin{aligned}
& a \oplus (a \oplus wa) && \\
& = (a \oplus a) \oplus (a \oplus wa), && \text{by idempotency of } \oplus, \\
& = (a \oplus a) \oplus (a \oplus wa), && \text{by idempotency of } \oplus, \\
& = (a \oplus a) \oplus (b \oplus wb), && \text{by the inductive hypothesis,} \\
& = (a \oplus b) \oplus (a \oplus wb), && \text{by transpositionality of } \oplus, \\
& = (a \oplus b) \oplus (a \oplus wa), && \text{by the inductive hypothesis,} \\
& = (a \oplus b) \oplus (b \oplus wb), && \text{by the inductive hypothesis,} \\
& = (b \oplus a) \oplus (b \oplus wb), && \text{by commutativity of } \oplus, \\
& = (b \oplus b) \oplus (a \oplus wb), && \text{by transpositionality of } \oplus, \\
& = (b \oplus b) \oplus (a \oplus wa), && \text{by the inductive hypothesis,} \\
& = (b \oplus b) \oplus (b \oplus wb), && \text{by the inductive hypothesis,} \\
& = b \oplus (b \oplus wb), && \text{by idempotency of } \oplus.
\end{aligned}$$

The converse to the above is rather more complicated; we give the idea below.

Theorem 5.1.19. [ᚠ] *A cancellative midpoint algebra* $(A, \oplus) \colon \texttt{Midpoint-algebra}$ *that is iterative by* $M \colon A^{\mathbb{N}} \to A$ *has finite approximations.*

Proof (Sketch). [f] We first prove the *finite-cancellation property*: that for any $w, z \colon A$ and $\alpha \colon A^{\mathbb{N}}$, if for any $n \colon \mathbb{N}$ we have $\alpha_0 \oplus (\alpha_1 \oplus ...(\alpha_{n-1} \oplus w)) = \alpha_0 \oplus (\alpha_1 \oplus ...(\alpha_{n-1} \oplus z))$, then $w = z$. The proof of this is straightforward by the fact $\oplus$ is cancellative and induction on $n\colon \mathbb{N}$.

We next prove *one-sided approximation*: that for any $a\colon A$ and $\theta\colon A^{\mathbb{N}}$, if we have a sequence $\zeta\colon A^{\mathbb{N}}$ such that $a = \theta_0 \oplus (\theta_1 \oplus \ldots(\theta_{n-1} \oplus \zeta_n))$ for all $n\colon \mathbb{N}$, then $a = M(\theta)$. The proof of this is by the second iteration sub-property (Definition 5.1.7), as if we set $\alpha := \theta$, $\beta_0 := a$ and $\beta_{i+1} = \zeta_{i+1}$ then once we prove the antecedent — that for all $i\colon \mathbb{N}$ we have $\beta_i = \alpha_i \oplus \beta_{i+1}$ — the result will follow. The antecedent follows by induction on i. In the base case, where $i := 0$, we give $a = \theta_0 \oplus \zeta_1$ by the above equation concerning α and θ when $n := 1$. In the inductive case, where $i := i' + 1$ for some $i'\colon \mathbb{N}$, we give $\zeta_i = \theta_i \oplus \zeta_{i+1}$ by the equation when $n := i + 1$ and the finite-cancellation property.

Finally, we prove that we have *finite approximations*: that for all $\alpha, \beta\colon A^{\mathbb{N}}$, if we have sequences $\theta, \zeta\colon A^{\mathbb{N}}$ such that $\alpha_0 \oplus (\alpha_1 \oplus \ldots(\alpha_{n-1} \oplus \theta_n)) = \beta_0 \oplus (\beta_1 \oplus \ldots(\beta_{n-1} \oplus \zeta_n))$ for all $n\colon \mathbb{N}$, then $M(\alpha) = M(\beta)$. This follows by cancellation once we prove that $M(\alpha) \oplus M(\mathsf{tail}\ \theta) = M(\beta) \oplus M(\mathsf{tail}\ \theta)$, and in turn this follows by the homomorphic property (Lemma 5.1.12) once we prove that $M(\lambda i.\alpha_i \oplus \theta_{i+1}) = M(\lambda i.\beta_i \oplus \theta_{i+1})$. We prove this using one-sided approximation: by setting $\beta' := \lambda i.\beta_i \oplus \theta_{i+1}$ the result is reduced to showing that there is a sequence $\gamma\colon A^{\mathbb{N}}$ such that $M(\lambda i.\alpha_i \oplus \theta_{i+1}) = \beta'_0 \oplus (\beta'_1 \oplus \ldots(\beta'_{n-1} \oplus \gamma_n))$ for all $n\colon \mathbb{N}$.

This sequence is defined $\gamma_n := M((\beta_n \oplus \zeta_{n+1}) :: (\lambda i.\alpha_{i+n+1} \oplus \theta_{i+n+2}))$. Given any $n\colon \mathbb{N}$, we prove that $M(\lambda i.\alpha_i \oplus \theta_{i+1}) = (\beta_0 \oplus \theta_1) \oplus ((\beta_1 \oplus \theta_2) \oplus \ldots((\beta_{n-1} \oplus \theta_n) \oplus M((\beta_n \oplus \zeta_{n+1}) :: (\lambda i.\alpha_{i+n+1} \oplus \theta_{i+n+2}))))$ by various straightforward rearrangements[a] of $\oplus$ and M, and by the assumption that $\alpha_0 \oplus (\alpha_1 \oplus \ldots(\alpha_{n-1} \oplus \theta_n)) = \beta_0 \oplus (\beta_1 \oplus \ldots(\beta_{n-1} \oplus \zeta_n))$.

[a]For specific details on these rearrangements, we invite the interested reader to view the formalisation.

Corollary 5.1.20. [f] *Given a midpoint algebra $(A, \oplus)$, some type X and two functions $f, g\colon X \to A^{\mathbb{N}}$, if for all $x\colon X$ and $n\colon \mathbb{N}$ we have that $f(x)$ and $g(x)$ are n-approximately equal, then $(M \circ f) : X \to A$ and $(M \circ g) : X \to A$ are pointwise-equal.*

Proof. [f] By Theorem 5.1.19.

5.1.4 Bipointed convex bodies

Adding iteration to a cancellative midpoint algebra gives us the structure we call an *abstract convex body*.

Definition 5.1.21. [ᚠ] A *convex body* is a cancellative midpoint algebra $(A, \oplus)$ that is iterative by $M : A^{\mathbb{N}} \longrightarrow A$. For such a structure, we write $(A, \oplus, M) : \text{Convex-body}$ with the M operator explicit and the proof terms implicit.

Every line segment of $\mathbb{R}^n$ is an abstract convex body; following this fashion, a closed line segment corresponds to a *bipointed convex body* [ES01].

Definition 5.1.22. A *bipointed convex body* is a convex body $(A, \oplus, M)$ with two distinguished points (the '*endpoints*') $u, v : A$,

$$\text{Bipointed-convex-body} := \sum_{((A,\oplus,M) : \text{Convex-body})} (A \times A) .$$

Note that the above defines the *type of* bipointed convex bodies.

Finally, a closed and bounded line segment – called an *interval object* – is defined as a bipointed convex body that satisfies the following universal property.

Definition 5.1.23. [ᚠ] A bipointed convex body $(A, \oplus_A, M_A, u, v)$ satisfies the *universal property of interval objects* if, given any bipointed convex body $(B, \oplus_B, M_B, s, t)$ there is a unique[a] function $h : A \longrightarrow B$ that maps the endpoints of A to their respective endpoints of B and is a midpoint homomorphism,

$$\text{is-interval-object}((A, \oplus_A, M_A, u, v)) := \prod_{((B,\oplus_B,M_B,s,t) : \text{Bipointed-convex-body})}$$

$$\exists!_{(h : A \to B)} \; ((h(u) = s) \times (h(v) = t) \times \text{is-midpoint-hom}((A, \oplus_A), (B, \oplus_B), h)) .$$

[a] Note we use here the TypeTopology notation for Σ-types that have a unique witness. Given a type $X : \mathcal{U}$ and $Y : X \longrightarrow V$, *unique existence* is the type that proves ΣY is a singleton: i.e. $\exists!_{(A : X \to \mathcal{V})} := \sum_{(p : \Sigma A)} \prod_{(q : \Sigma A)} (p = q)$.

The idea of the universal property is illustrated in Figure 5.1.

Definition 5.1.24. [ᚠ] An *interval object* is a bipointed convex body that satisfies the universal property of interval objects,

$$\text{interval-object}_{\mathcal{U}} := \sum_{((A,\oplus,M,u,v) : \text{Bipointed-convex-body})} \text{is-interval-object}((A, \oplus, M, u, v)).$$

For such a structure, we write $(A, \oplus, M, u, v) : \text{interval-object}$ with the proof term of the universal property implicit.

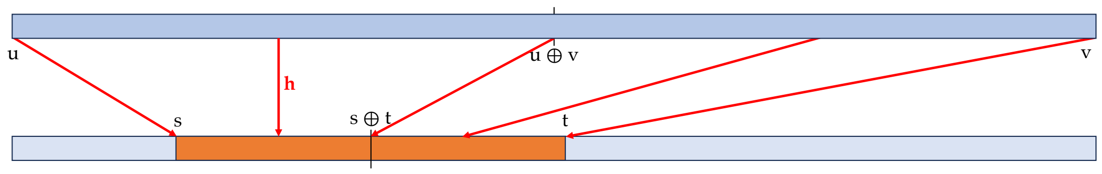


Figure 5.1: Illustration of the universal property Definition 5.1.23; the map $h\colon A \longrightarrow B$ maps points on the bipointed convex body $(A, \oplus_A, M_A, u, v)$ to the relative point on the bipointed convex body $(B, \oplus_B, M_B, s, t)$. The colours used here are as in the illustration.

Note that the above defines the *type of* interval objects. This type is in fact a subsingleton in any univalent universe: we proved this using Escardó's TypeTopology version of the *structure identity principle* [Esc20; ANST20].

Theorem 5.1.25. [◎] *For any universe* $\mathcal{U}$ *such that* is-univalent$(\mathcal{U})$, *it is the case that* is-prop(interval-object$_{\mathcal{U}}$).

Proof (Sketch). [fu] By the universal property (Definition 5.1.23), any two interval objects are equivalent — thus, by univalence, they are equal. For the full proof, see [Amb20c].

Given this, we will only ever want to use the map derived from the universal property from an interval object to cast objects from and to the same underlying convex body.

We define, using the universal property map, the following *affine maps* on an interval object, which cast objects from that interval object onto "sub-intervals" of that object.

Definition 5.1.26. [Ɣ] Given an interval object $(A, \oplus, M, u, v)$ we define the *affine map* affine: $A \longrightarrow A \longrightarrow A \longrightarrow A$ as,

$$\mathsf{affine}(s, t, a) := h(a),$$

where $h\colon A \longrightarrow A$ is the map derived from the universal property, which maps elements from the interval object to the 'sub-interval' bipointed convex body $(A, \oplus, M, s, t)$.

Lemma 5.1.27. [Ɣ] [Ɣ] *Given an interval object* $(A, \oplus, M, u, v)$, *the affine map* affine$(s, t)\colon A \longrightarrow A$, *for any points* $s, t\colon A$, *correctly maps the endpoints; i.e.* affine$(s, t, u) = s$ *and* affine$(s, t, v) = t$.

Proof. By the correct mapping of endpoints requirement in Definition 5.1.23.

Lemma 5.1.28. [Ɏ] *Given an interval object $(A, \oplus, M, u, v)$, the affine map* $\mathsf{affine}(s,t) \colon A \longrightarrow A$*, for any points $s, t \colon A$, is a midpoint homomorphism,*

$$\text{is-midpoint-hom}((A, \oplus), \mathsf{affine}(s,t)).$$

Proof. By the midpoint homomorphism requirement in Definition 5.1.23.

Corollary 5.1.29. [Ɏ] *Given an interval object $(A, \oplus, M, u, v)$, the affine map* $\mathsf{affine}(s,t) \colon A \longrightarrow A$*, for any points $s, t \colon A$, is an M-homomorphism,*

$$\text{is-M-hom}((A, \oplus), M, \mathsf{affine}(s,t)).$$

Proof. By Lemmas 5.1.15 and 5.1.28.

Many properties of our later-defined computational functions are proved by the following lemma, which gives the uniqueness of the affine map.

Lemma 5.1.30. [Ɏ] *Given an interval object $(A, \oplus, M, u, v)$, the affine map* $\mathsf{affine}(s,t) \colon A \longrightarrow A$*, for any points $s, t \colon A$, is identical to any function $f \colon A \longrightarrow A$ that satisfies (i) $f(u) = s$, (ii) $f(v) = t$ and (iii)* $\text{is-midpoint-hom}(f)$.

Proof. By the uniqueness requirement in Definition 5.1.23.

The affine map functional is the computational seed of the interval object, forth from which springs the algorithms we will define on the interval, which represent algorithms on the real numbers. Two basic examples are the identity function and any constant map.

Lemma 5.1.31. [Ɏ] *Given an interval object $(A, \oplus, M, u, v)$, the identity function* $\mathsf{id} \colon A \longrightarrow A$ *is given by the affine map* $\mathsf{affine}(u,v) \colon A \longrightarrow A$.

Proof. The function $\mathsf{id} \colon A \longrightarrow A$ trivially satisfies (i) $\mathsf{id}(u) = u$, (ii) $\mathsf{id}(v) = v$ and (iii) $\prod_{(a,b \colon A)} (\mathsf{id}(a \oplus b) = \mathsf{id}(a) \oplus \mathsf{id}(b))$. Therefore, by Lemma 5.1.30, $\mathsf{id} = \mathsf{affine}(u,v)$.

Lemma 5.1.32. [Ɏ] *Given an interval object $(A, \oplus, M, u, v)$, the constant function $(\lambda(- \colon A).x) \colon A \longrightarrow A$, for any $x \colon A$, is given by the affine map* $\mathsf{affine}(x,x) \colon A \longrightarrow A$.

Proof. For any $x : A$ the function $(\lambda(- : A).x) : A \to A$ trivially maps both endpoints of A to the only endpoint x of the sub-interval, and is a midpoint homomorphism by idempotency of $\oplus$ in Definition 5.1.2. Therefore, by Lemma 5.1.30, $(\lambda(- : A).x) = \mathsf{affine}(x, x)$.

5.1.5 Arithmetic on $[-1, 1]$ by assuming an interval object

For the rest of this section, we assume that an interval object $\mathbb{I} : \mathsf{interval\text{-}object}_{\mathcal{U}}$ is given in any given type universe $\mathcal{U}$ — we will *not* try to construct the interval object in this thesis.

Assumption 5.1.33. [⚑] We assume the existence of $\mathbb{I} := (\mathbb{I}, \oplus, M, -1, +1)$ to specify the closed real interval $[-1, 1]$.

Note that we abuse notation and write $\mathbb{I}$ both for the interval object itself and the set over which it operates.

Of course $-1, +1 : \mathbb{I}$ represent those same numbers. Using the midpoint, we can also represent other dyadic numbers in this interval.

Definition 5.1.34. [⚑] The distinguished element $0 : \mathbb{I}$ is given by $-1 \oplus +1$.

Arithmetic functions such as negation and multiplication are not further axioms of this specification. They are instead defined, and their properties derived, from the existing — rather small number — of axioms in the interval object specification. In particular, as we saw with the two trivial examples above, the affine map is used to define both negation and multiplication.

Negation

Negation is defined by using the affine map that ‘flips’ the interval, casting elements of $\mathbb{I}$ to their negated counterparts.

Definition 5.1.35. [⚑] *Negation* is defined on the interval $\mathbb{I}$ as the unique function $- : \mathbb{I} \to \mathbb{I}$,

$$-x := \mathsf{affine}(+1, -1, x).$$

Lemma 5.1.36. [⚑] [⚑] [⚑] *Negation is the unique function on $\mathbb{I}$ that negates every element of the object, i.e.,*

(i) $-(-1) = +1$,

(ii) $-(+1) = -1$,

(iii) $\prod_{(x,y : \mathbb{I})} (-(x \oplus y) = -x \oplus -y)$.

Proof. By Lemmas 5.1.27, 5.1.28 and 5.1.30.

We can further show that negation behaves as we expect; for example by showing it has a unit and is involutive.

Corollary 5.1.37. [⚑] *Negating* $0 : \mathbb{I}$ *gives back* 0.

Proof. By definition $0 := -1 \oplus +1$, so we wish to show that $-(-1 \oplus +1) = -1 \oplus +1$. By Lemma 5.1.36, $-(-1 \oplus +1) = -(-1) \oplus -(+1)$ because negation is a midpoint homomorphism; then, because negation negates the endpoints, we have $-(-1) \oplus -(+1) = +1 \oplus -1$. The final step $+1 \oplus -1 = -1 \oplus +1$ follows by the commutativity of the midpoint operator in Definition 5.1.2.

Lemma 5.1.38. [⚑] *Negation on* $\mathbb{I}$ *is an involution, i.e.* $-(-x) = x$ *for all* $x : \mathbb{I}$.

Proof. We first use the fact that negation negates the endpoints (Lemma 5.1.36) to derive $- - (-1) = -(+1) = -1$ and $- - (+1) = -(-1) = +1$. Also, because negation is a midpoint homomorphism (Lemma 5.1.36) and the composition of two homomorphisms is itself a homomorphism (Lemma 5.1.4), double negation is a midpoint homomorphism. Therefore, because any affine map is unique (Lemma 5.1.30) and double negation is a midpoint homomorphism that maps the endpoints of $\mathbb{I}$ to themselves, it is the case that $(\lambda(x : \mathbb{I}). -(-x)) = \mathsf{affine}(-1, +1)$. The proof follows by the fact that, due to its identification with $\mathsf{affine}(-1, +1)$, double negation gives the identity map (Lemma 5.1.31) and therefore $-(-x) = \mathsf{id}(x) = x$ for all $x : \mathbb{I}$.

Multiplication

Multiplication by some $x : \mathbb{I}$ is defined by using the affine map that maps the interval to the sub-interval $[-x, x]$.

Definition 5.1.39. [⚑] *Multiplication* is defined on the interval $\mathbb{I}$ as the unique function $* : \mathbb{I} \to \mathbb{I} \to \mathbb{I}$,

$$x * y := \mathsf{affine}(-x, x, y).$$

Lemma 5.1.40. [ᚠ] [ᚠ] [ᚠ] *Multiplication of some* $x : 𝕀$ *is the unique function on* $𝕀$ *that multiplies elements of the object by* x*, i.e.,*

(i) $x * -1 = -x$,

(ii) $x * +1 = x$,

(iii) $\prod_{(x,y\,:\,𝕀)} (x * (y \oplus z) = (x * y) \oplus (x * z))$.

Proof. By Lemmas 5.1.27, 5.1.28 and 5.1.30.

Furthermore, multiplication also behaves as expected in a variety of ways.

Lemma 5.1.41. [ᚠ] *Multiplication on* $𝕀$ *satisfies* $-1 * y = -y$.

Proof. The function $\lambda y.{-1}{*}y :=$ affine$(-(-1), -1)$. By Lemma 5.1.36, this is pointwise-equal to affine$(+1, -1)$, which is the negation function.

Lemma 5.1.42. [ᚠ] *Multiplication on* $𝕀$ *satisfies* $+1 * y = y$.

Proof. The function $\lambda y.{+1}{*}y :=$ affine$(-(+1), +1)$. By Lemma 5.1.36, this is pointwise-equal to affine$(-1, +1)$, which by Lemma 5.1.31 is the identity function.

Lemma 5.1.43. [ᚠ] [ᚠ] *Multiplication on* $𝕀$ *satisfies* $x * 0 = 0$ *and* $0 * y = 0$.

Proof. The former follows by Lemma 5.1.40. The latter follows because the function $\lambda y.0{*}y :=$ affine$(-0, 0)$, and by Corollary 5.1.37 this is pointwise-equal to affine$(0, 0)$; this, by Lemma 5.1.32, is the constant function that outputs 0.

Theorem 5.1.44. [ᚠ] *Multiplication on* $𝕀$ *is commutative.*

Proof. Given any $x, y : 𝕀$, we wish to show that $x * y = y * x$; i.e. that affine$(-x, x, y) = y * x$. This is proved by showing affine$(-x, x) = \lambda y.y * x)$ by Lemma 5.1.30. Therefore, we must show that (1) $-1 * x = -x$, (2) $+1 * x = x$ and (3) is-midpoint-hom$(\lambda y.y * x)$.

The first two conditions are given by Lemmas 5.1.41 and 5.1.42. The third is more complicated: given any $y', z' : 𝕀$ we need to show that $(y' \oplus z') * x = (y' * x) \oplus (z' * x)$. This is given by another application of Lemma 5.1.30 wherein we show affine$(-(y' \oplus z'), y' \oplus z') = \lambda x.(y' * x) \oplus (z' * x)$. Therefore, we must now show that (1) $(y' * -1) \oplus (z' * -1) = -(y' \oplus z')$, (2) $(y' * +1) \oplus (z' * +1) = (y' \oplus z')$ and (3) is-midpoint-hom$(\lambda x.(y' * x) \oplus (z' * x)$.

For the first condition, by Lemma 5.1.40.(i) and (iii) we have that $(y' * -1) \oplus (z' * -1) = (-y' \oplus -z') = -(y' \oplus z')$. For the second, by Lemma 5.1.40.(ii) we have that $(y' * +1) \oplus (z' * +1) = (y' \oplus z')$. The third is again more complicated: given any $a, b : ▯$, the result is given by the following equational reasoning:

$$\begin{aligned} & (x * (a \oplus b)) \oplus (y * (a \oplus b)), \\ = & ((x * a) \oplus (x * b)) \oplus (y * (a \oplus b)) && \text{by Lemma 5.1.40.(iii),} \\ = & ((x * a) \oplus (x * b)) \oplus ((y * a) \oplus (y * b)) && \text{by Lemma 5.1.40.(iii),} \\ = & ((x * a) \oplus (y * a)) \oplus ((x * b) \oplus (y * b)) && \text{by trans. of } \oplus \text{ (Definition 5.1.2).} \end{aligned}$$

Lemma 5.1.45. [ᚠ] *Multiplication on ▯ satisfies* $((x \oplus y) * z) = (x * z) \oplus (y * z)$.

Proof. By Lemma 5.1.40 and Theorem 5.1.44.

Theorem 5.1.46. [ᚠ] *Multiplication on ▯ is associative.*

The proof that multiplication is associative has a similar proof technique as the proof of commutativity (Theorem 5.1.44). For the full details, we invite the interested reader to view the formalisation.

5.2 Verified ternary signed-digit encodings

5.2.1 Background and definition in our type theory

Boehm et al.'s early paper on exact real arithmetic explores multiple representations of real numbers, one of which is an approach where a real number is represented by an infinitary sequence of digits [BCRO86].

The idea is that a (base 2) *digit encoding* of a real number in the compact interval $[0, d]$ (for some $d : \mathbb{N}$) is an infinitary seqeuence of digits $\alpha : \{0, ..., d\}^{\mathbb{N}}$. The real number represented by such a digit encoding $[\![\alpha]\!] : [0, d]$ is defined by,

$$[\![\alpha]\!] := \sum_{n=0}^{\infty} \frac{\alpha_i}{2^{n+1}}.$$

For example, with $d := 2$, $[\![\{0, 2, 0, 2, 0, ...\}]\!] = 0.666...$ and $[\![\{0, 1, 0, 1, 0, ...\}]\!] = 0.333....$ This approach is problematic, however, as even addition is not definable [BCRO86]. An illuminating example is the addition of the two above realisers of 0.666... and

0.333.... Computing only the first digit of the output requires the function to look at an infinite number of digits of the input, because (for example) $[\![\{0, 2, 0, 2, 0, \ldots\}]\!] + [\![\{0, 1, 0, 1, 0, \ldots, 0, 0, 0, \ldots\}]\!] = 0.9\ldots$, whereas $[\![\{0, 2, 0, 2, 0, \ldots\}]\!] + [\![\{0, 1, 0, 1, 0, \ldots, 0, 2, 0, \ldots\}]\!] = 1.0\ldots$.

However, the *signed-digit encoding* representation does not have this problem. A signed-digit encoding of a real number in a compact interval $[-d, d]$ (for some $d\colon \mathbb{N}$) is an infinitary sequence of digits $\alpha\colon \{-d, \ldots, d\}^{\mathbb{N}}$. The real number represented by such a signed-digit encoding $[\![\alpha]\!]\colon \mathbb{R}$ is defined by,

$$[\![\alpha]\!] := \sum_{n=0}^{\infty} \frac{\alpha_n}{2^{n+1}},$$

which can be infinitely unfolded using a midpoint operator $x \oplus y := \frac{x+y}{2}$:

$$[\![\alpha]\!] := \alpha_0 \oplus (\alpha_1 \oplus (\alpha_2 \oplus \ldots)).$$

This representation of the reals uses extra 'redundant' digits than is necessary for representing each number in the interval. The redundant digits, however, are what enable addition (and multiplication) to be defined at the expense of having multiple representations for the same real number. In the previous addition example, the second item of the sequence could have been set as 2 and then, if necessary, corrected by negative digits later [Plu98] [Esc11b].

In this section, we recall the type of *ternary signed-digit encodings* of real numbers in the compact interval $[-1, 1]$, as extensively explored in the literature of exact real arithmetic [Di 93; Plu98; Ber09]. Every real number in $[-1, 1]$ can be represented by a function of type $\mathbb{N} \to \{-1, 1\}$, but by using the redundant representation $\mathbb{N} \to \{-1, 0, 1\}$ we can perform exact real arithmetic on representations of this compact interval.

Our formalisation is based on Escardó's Haskell library for exact real computation on ternary signed-digits [Esc11b]. We first convert his definitions of various arithmetic functions to Agda, which is non-trivial as we must convince Agda's termination checker. Following this, we verify the correctness of his algorithms using the interval object seen in Section 5.1, showing that they encode the correct operations on the real numbers.

Definition 5.2.1. [⚐] The type of *ternary digits* $\mathbb{3}$, equivalent to $\mathsf{Fin}(3)$, is defined by its elements $\overline{1}, 0, 1\colon \mathbb{3}$.

Definition 5.2.2. [ᚠ] A *ternary signed-digit encoding* of a real number in $[-1, 1]$ is any function $\alpha \colon \mathbb{3}^{\mathbb{N}}$.

5.2.2 Representation via the interval object

A *representation map* takes a representation of a real to the real that it represents.

Definition 5.2.3. [ᚠ] [ᚠ] [ᚠ] Given a type of real numbers $\mathbb{R}$, a type for representing reals K and a representation map $[\![-]\!] \colon K \to \mathbb{R}$, the n-ary function $\overline{f} \colon K^n \to K$ (for $n \colon \mathbb{N}$) *realises* a function $f \colon \mathbb{R}^n \to \mathbb{R}$ if,

$$\prod_{((x_0,\ldots,x_{n-1}) \colon K^n)} [\![\overline{f}\,(x_0, \ldots, x_{n-1})]\!] = f\left([\![x_0]\!], \ldots, [\![x_{n-1}]\!]\right).$$

The idea here is that in order to verify $\overline{f} \colon K^n \to K$ realises $f \colon \mathbb{R} \to \mathbb{R}$, using the representation map $[\![-]\!] \colon K \to \mathbb{R}$, we show that the diagram in Figure 5.2 commutes.

$$\begin{tikzcd}
K^n \arrow[r, "\overline{f}"] \arrow[d, "{[\![-]\!] \times \ldots \times [\![-]\!]}"'] & K \arrow[d, "{[\![-]\!]}"] \\
\mathbb{R}^n \arrow[r, "f"'] & \mathbb{R}
\end{tikzcd}$$

Figure 5.2: Commutative diagram illustrating Definition 5.2.3.

The identity map always has a realiser, and the composition of two realisers is a realiser of the composition of the two realised functions; i.e. the diagrams in Figure 5.3 commute.

$$\begin{tikzcd}
K \arrow[r, "\mathrm{id}_K"] \arrow[d, "{[\![-]\!]}"'] & K \arrow[d, "{[\![-]\!]}"] \\
\mathbb{R} \arrow[r, "\mathrm{id}_{\mathbb{R}}"'] & \mathbb{R}
\end{tikzcd}
\qquad
\begin{tikzcd}
K \arrow[r, "\overline{f}"] \arrow[d, "{[\![-]\!]}"'] & K \arrow[r, "\overline{g}"] \arrow[d, "{[\![-]\!]}"] & K \arrow[d, "{[\![-]\!]}"] \\
\mathbb{R} \arrow[r, "f"'] & \mathbb{R} \arrow[r, "g"'] & \mathbb{R}
\end{tikzcd}$$

Figure 5.3: Commutative diagrams illustrating Lemma 5.2.4.

Lemma 5.2.4. [ᚠ] [ᚠ] *The identity map is realised by the identity map, and composition preserves realisers.*

Proof. For the identity map, we need to show that for all $k\colon K$ we have $[\![\mathrm{id}_K(k)]\!] = \mathrm{id}_{\mathbb{R}}([\![k]\!])$; this is clearly the case by reflexivity. For composition of functions $\overline{f}, \overline{g}\colon K \longrightarrow K$, which realise $f, g\colon \mathbb{R} \longrightarrow \mathbb{R}$ respectively, we want to show that $[\![\overline{g}(\overline{f}(k))]\!] = g(f([\![k]\!]))$. This is the case because $[\![\overline{g}(\overline{f}(k))]\!] = g([\![\overline{f}(x)]\!])$ (because $\overline{f}$ realises f) and then $g([\![\overline{f}(x)]\!]) = g(f([\![k]\!]))$ (because $\overline{g}$ realises g).

We define the *representation map* for ternary signed-digit encodings, which maps such encodings of reals in $[-1, 1]$ to the numbers they represent on the interval object $\mathbb{I}$, using the infinitary midpoint operator $M\colon \mathbb{I}^{\mathbb{N}} \longrightarrow \mathbb{I}$.

Definition 5.2.5. [ᚠ] We define the *representation map* from ternary digits $\mathbb{3}$ to the interval object $\mathbb{I}$,

$$\begin{aligned} \langle - \rangle &: \mathbb{3} \longrightarrow \mathbb{I}, \\ \langle \overline{1} \rangle &:= -1, \\ \langle 0 \rangle &:= 0, \\ \langle 1 \rangle &:= +1, \end{aligned}$$

Definition 5.2.6. [ᚠ] We define the *representation map* from ternary signed-digit encodings $\mathbb{3}^{\mathbb{N}}$ to the interval object $\mathbb{I}$,

$$\begin{aligned} \langle\!\langle - \rangle\!\rangle &: \mathbb{3}^{\mathbb{N}} \longrightarrow \mathbb{I}, \\ \langle\!\langle \alpha \rangle\!\rangle &:= M(\mathsf{map}(\langle - \rangle, \alpha)). \end{aligned}$$

In order to verify that an operation on ternary digits $\overline{f'}\colon \mathbb{3}^n \longrightarrow \mathbb{3}$ or one on ternary signed-digit encodings $\overline{f}\colon (\mathbb{3}^{\mathbb{N}})^n \longrightarrow \mathbb{3}^{\mathbb{N}}$ correctly realises an operation on the interval object $f\colon \mathbb{I}^n \longrightarrow \mathbb{I}$, we will show that the diagrams in Figure 5.4 commute.

$$\begin{tikzcd} \mathbb{3}^n \arrow[r, "\overline{f'}"] \arrow[d, "\langle - \rangle^n"'] & \mathbb{3} \arrow[d, "\langle - \rangle"] \\ \mathbb{I}^n \arrow[r, "f"'] & \mathbb{I} \end{tikzcd} \qquad \begin{tikzcd} (\mathbb{3}^{\mathbb{N}})^n \arrow[r, "\overline{f}"] \arrow[d, "\langle\!\langle - \rangle\!\rangle^n"'] & \mathbb{3}^{\mathbb{N}} \arrow[d, "\langle\!\langle - \rangle\!\rangle"] \\ \mathbb{I}^n \arrow[r, "f"'] & \mathbb{I} \end{tikzcd}$$

Figure 5.4: Commutative diagrams illustrating Definition 5.2.3 (which is itself illustrated in Figure 5.2) using the representation maps defined in Definition 5.2.5 (left) and Definition 5.2.6 (right).

5.2.3 Exact real arithmetic

Negation

Negation on signed-digit encodings is straightforward to define — we simply flip every digit of the sequence.

Definition 5.2.7. [ ] [ ] We first define the negation function flip on ternary digits in the expected way:

$$\begin{aligned}
&\mathsf{flip} : \mathbb{3} \to \mathbb{3},\\
&\mathsf{flip}(\overline{1}) := 1,\\
&\mathsf{flip}(0) := 0,\\
&\mathsf{flip}(1) := \overline{1}.
\end{aligned}$$

Then, the negation function neg on ternary signed-digit encodings is defined as follows:

$$\begin{aligned}
&\mathsf{neg} : \mathbb{3}^{\mathbb{N}} \to \mathbb{3}^{\mathbb{N}},\\
&\mathsf{neg}(x) \quad := \mathsf{map}(\mathsf{flip}, x).
\end{aligned}$$

We then verify these operations; i.e. we show that the following diagrams commute.

$$\begin{tikzcd}
\mathbb{3} \arrow[r, "\mathsf{flip}"] \arrow[d, "\langle - \rangle"'] & \mathbb{3} \arrow[d, "\langle - \rangle"] & \mathbb{3}^{\mathbb{N}} \arrow[r, "\mathsf{neg}"] \arrow[d, "\langle - \rangle"'] & \mathbb{3}^{\mathbb{N}} \arrow[d, "\langle - \rangle"] \\
\mathbb{I} \arrow[r, "-"'] & \mathbb{I} & \mathbb{I} \arrow[r, "\oplus"'] & \mathbb{I}
\end{tikzcd}$$

Figure 5.5: Commutative diagrams illustrating Lemma 5.2.8 (left) and Theorem 5.2.10 (right).

Lemma 5.2.8. [ ] *The negation function on ternary digits realises the negation function on the interval object.*

Proof. We prove $\langle \mathsf{flip}(t) \rangle = -\langle t \rangle$ by case splitting on the given $t : \mathbb{3}$. In the $t := 0$ case, we show $\langle 0 \rangle = -\langle 0 \rangle$ by Corollary 5.1.37. In the other two cases, the proof follows by the mapping of the endpoints in Lemma 5.1.36.

In order to verify negation on signed-digit encodings, we use the fact that the map function preserves a realiser; this is illustrated by the diagram in Figure 5.6.

$$\begin{tikzcd} \mathbb{3} \arrow[r, "\overline{f}"] \arrow[d, "\langle - \rangle"'] & \mathbb{3} \arrow[d, "\langle - \rangle"] \\ \mathbb{I} \arrow[r, "f"'] & \mathbb{I} \end{tikzcd} \Longrightarrow \begin{tikzcd} \mathbb{3}^{\mathbb{N}} \arrow[r, "\mathsf{map}\ \overline{f}"] \arrow[d, "\langle\!\langle - \rangle\!\rangle"'] & \mathbb{3}^{\mathbb{N}} \arrow[d, "\langle\!\langle - \rangle\!\rangle"] \\ \mathbb{I} \arrow[r, "f"'] & \mathbb{I} \end{tikzcd}$$

Figure 5.6: Commutative diagram illustrating Lemma 5.2.9.

Lemma 5.2.9. [ ] *If a function* $\overline{f} : \mathbb{3} \to \mathbb{3}$ *on ternary digits realises a midpoint homomorphism* $f : \mathbb{I} \to \mathbb{I}$ *on the interval object, then* $\mathsf{map}(\overline{f}) : \mathbb{3}^{\mathbb{N}} \to \mathbb{3}^{\mathbb{N}}$ *realises* f *on signed-digit encodings.*

Proof. [f] We want to show that, for all $\alpha : \mathbb{3}^{\mathbb{N}}$, $\langle\!\langle \mathsf{map}(\overline{f}, \alpha) \rangle\!\rangle = f(\langle\!\langle \alpha \rangle\!\rangle)$. By function extensionality and Definition 5.2.6, the left-hand side of the equation becomes $M(\lambda n.\langle \overline{f}(\alpha_n) \rangle)$. By the fact that $\overline{f}$ realises f, it further becomes $M(\lambda n.f(\langle \alpha_n \rangle))$. By Lemma 5.1.15, f is an M-homomorphism, and therefore the equation becomes $f(M(\lambda n.\langle \alpha_n \rangle))$; which is definitionally equal to the conclusion by Definition 5.2.6.

Theorem 5.2.10. [ ] *The negation function on signed-digit encodings realises the negation function on the interval object.*

Proof. [f] By Lemmas 5.2.8, 5.2.9 and 5.1.36.

Binary midpoint

The midpoint functions — both binary and infinitary, which are intended to realise $\oplus$ and M, respectively — are much more complicated to define on signed-digit encodings. We follow the definitions Escardó gave in [Esc11b], formalising them in Agda for our framework.

The binary midpoint defined here sums two encodings of type $\mathbb{3}^{\mathbb{N}}$ (which encodes the interval $[-1, 1]$) to achieve an encoding of type $\mathbb{5}^{\mathbb{N}}$ (which encodes the interval $[-2, 2]$) which is then divided by two to output an encoding once again in $\mathbb{3}^{\mathbb{N}}$.

Definition 5.2.11. [ ] The type of *quinary digits* $\mathbb{5}$, equivalent to $\mathsf{Fin}(5)$, is defined by its elements $\overline{2}, \overline{1}, 0, 1, 2 : \mathbb{5}$.

Definition 5.2.12. [ ] We define the addition function on ternary digits $\mathrm{add}\mathbb{3} \colon \mathbb{3} \to \mathbb{3} \to \mathbb{5}$ in the expected way by pattern matching (e.g. $\mathrm{add}\mathbb{3}(\overline{1}, \overline{1}) := \overline{2}$), etc.).

Now we define the halving function $\mathrm{div2} \colon \mathbb{5}^{\mathbb{N}} \to \mathbb{3}^{\mathbb{N}}$ corecursively (though, as we see in the below Remark 5.2.14, we define the function by induction in our AGDA formalisation). When the first element is $\overline{2}, 0$ or 2, the output is straightforward. Otherwise, the definition must 'offset' the output using the redundant digits $\overline{1}$ and 1.

Definition 5.2.13. [ ] We define the halving function from quinary signed-digit encodings to ternary signed-digit encodings as,

$$
\begin{aligned}
&\mathrm{div2} : \mathbb{5}^{\mathbb{N}} \to \mathbb{3}^{\mathbb{N}},\\
&\mathrm{div2}(\overline{2} :: \alpha) := \overline{1} :: \mathrm{div2}(\alpha),\\
&\mathrm{div2}(\overline{1} :: \overline{2} :: \alpha) := \overline{1} :: \mathrm{div2}(0 :: \alpha),\\
&\mathrm{div2}(\overline{1} :: \overline{1} :: \alpha) := \overline{1} :: \mathrm{div2}(1 :: \alpha),\\
&\mathrm{div2}(\overline{1} :: 0 :: \alpha) := 0 :: \mathrm{div2}(\overline{2} :: \alpha),\\
&\mathrm{div2}(\overline{1} :: 1 :: \alpha) := 0 :: \mathrm{div2}(\overline{1} :: \alpha),\\
&\mathrm{div2}(\overline{1} :: 2 :: \alpha) := 0 :: \mathrm{div2}(0 :: \alpha),\\
&\mathrm{div2}(0 :: \alpha) := 0 :: \mathrm{div2}(\alpha),\\
&\mathrm{div2}(1 :: \overline{2} :: \alpha) := 0 :: \mathrm{div2}(0 :: \alpha),\\
&\mathrm{div2}(1 :: \overline{1} :: \alpha) := 0 :: \mathrm{div2}(1 :: \alpha),\\
&\mathrm{div2}(1 :: 0 :: \alpha) := 0 :: \mathrm{div2}(2 :: \alpha),\\
&\mathrm{div2}(1 :: 1 :: \alpha) := 1 :: \mathrm{div2}(\overline{1} :: \alpha),\\
&\mathrm{div2}(1 :: 2 :: \alpha) := 1 :: \mathrm{div2}(0 :: \alpha),\\
&\mathrm{div2}(2 :: \alpha) := 1 :: \mathrm{div2}(\alpha).
\end{aligned}
$$

Remark 5.2.14. [ ] We actually define this function in AGDA using an auxiliary function

$$\mathrm{div2}' \colon \mathbb{5} \times \mathbb{5} \to \mathbb{3} \times \mathbb{5}$$

such that we define

$$
\begin{aligned}
\mathrm{div2}(\alpha)_0 &:= \mathrm{pr}_1(\mathrm{div2}'(\alpha_0, \alpha_1)),\\
\mathrm{div2}(\alpha)_{n+1} &:= \mathrm{div2}(\mathrm{pr}_2(\mathrm{div2}'(\alpha_0, \alpha_1) :: \mathrm{tail}(\mathrm{tail}\ \alpha))).
\end{aligned}
$$

The values of $\mathrm{div2}'$ can be seen in Definition 5.2.13. For example $\mathrm{div2}'(\overline{1}, \overline{2}) := (\overline{1}, 0)$.

We use this function to define the binary midpoint function.

Definition 5.2.15. [ ] The binary midpoint function on ternary signed-digit encodings is defined by adding the two sequences and then halving the result:

$$\begin{aligned}
&\mathsf{mid} : \mathbb{3}^{\mathbb{N}} \to \mathbb{3}^{\mathbb{N}} \to \mathbb{3}^{\mathbb{N}},\\
&\mathsf{mid}(\alpha,\beta) \quad := \mathsf{div2}(\mathsf{zipWith}(\mathsf{add}\mathbb{3},\alpha,\beta)).
\end{aligned}$$

To verify that mid correctly realises $\oplus$, we utilise the following halving map half and prove its relationship to div2

Definition 5.2.16. [ ] We define the halving map from quinary digits to the interval object as,

$$\begin{aligned}
\mathsf{half} &: \mathbb{5} \to \mathbb{I},\\
\mathsf{half}(\overline{2}) &:= -1,\\
\mathsf{half}(\overline{1}) &:= -1 \oplus 0,\\
\mathsf{half}(0) &:= 0,\\
\mathsf{half}(1) &:= +1 \oplus 0,\\
\mathsf{half}(2) &:= +1.
\end{aligned}$$

$$\begin{tikzcd}
\mathbb{3}\times\mathbb{3} \arrow[r, "\mathsf{add}\mathbb{3}"] \arrow[d, "\langle - \rangle \times \langle - \rangle"'] & \mathbb{5} \arrow[d, "\mathsf{half}"] \\
\mathbb{I}\times\mathbb{I} \arrow[r, "\oplus"'] & \mathbb{I}
\end{tikzcd}
\qquad
\begin{tikzcd}
\mathbb{3}^{\mathbb{N}}\times\mathbb{3}^{\mathbb{N}} \arrow[r, "\mathsf{zipWith}(\mathsf{add}\mathbb{3})"] \arrow[rr, bend left, "\mathsf{mid}"] \arrow[d, "\langle - \rangle \times \langle - \rangle"'] & \mathbb{5}^{\mathbb{N}} \arrow[r, "\mathsf{div2}"] \arrow[d, "M\circ\mathsf{map}(\mathsf{half})"] & \mathbb{3}^{\mathbb{N}} \arrow[d, "\langle - \rangle"] \\
\mathbb{I}\times\mathbb{I} \arrow[r, "\oplus"'] & \mathbb{I} \arrow[r, equal] & \mathbb{I}
\end{tikzcd}$$

Figure 5.7: Commutative diagrams illustrating Lemma 5.2.19 (left) and Lemmas 5.2.18 and 5.2.20 and Theorem 5.2.21 (right).

We first prove the following lemma about the auxiliary function div2′ (Remark 5.2.14).

Lemma 5.2.17. [ ] *Given any $x, y\colon \mathbb{5}$ and $z\colon \mathbb{I}$, we have*

$$\langle a\rangle \oplus (\mathsf{half}\ b \oplus z) = (\mathsf{half}\ x \oplus (\mathsf{half}\ y \oplus z)),$$

where $a, b\colon \mathbb{3}\times\mathbb{5} := \mathsf{div2}'(\alpha_0,\alpha_1)$.

Proof. The cases where $a := \{\overline{2}, 0, 2\}$ are trivial. The ten other cases all require slightly different approaches, but all are just the rearrangement of elements of the midpoint algebra by idempotency, commutativity and transpositionality (Definition 5.1.2).

Lemma 5.2.18. [ϒ] *Given any quinary signed-digit encoding* $\alpha : \mathbb{5}^{\mathbb{N}}$, *we have that* $\langle\!\langle \mathsf{div2}(\alpha) \rangle\!\rangle = M(\mathsf{map}(\mathsf{half}, \alpha))$.

Proof. [f] By using Corollary 5.1.20, this is reduced to showing $\mathsf{map}(\langle - \rangle, \mathsf{div2}(\alpha))$ and $\mathsf{map}(\mathsf{half}, \alpha)$, for any $\alpha : \mathbb{5}^{\mathbb{N}}$, are n-approximately equal for any $n : \mathbb{N}$. In the base case, when $n := 0$, the proof is trivial — we can set $z, w : \mathbb{I}$ as any elements of $\mathbb{I}$, and so we only need to consider the inductive case where $n := n' + 1$.

Recall from Definition 5.1.16 that this means we must give some $z, w : \mathbb{I}$ that satisfy

$$\langle \mathsf{div2}(\alpha)_0 \rangle \oplus (\langle \mathsf{div2}(\alpha)_1 \rangle \oplus \ldots (\mathsf{div2}(\alpha)_n \oplus w)) = \mathsf{half}(x_0) \oplus (\mathsf{half}(\alpha_1) \oplus \ldots (\mathsf{half}(\alpha_n) \oplus z)).$$

By taking $(a, b) : \mathbb{3} \times \mathbb{5} := \mathsf{div2}'(\alpha_0, \alpha_1)$ this reduces to

$$\langle a \rangle \oplus (\langle \mathsf{div2}(b :: \alpha')_1 \rangle \oplus \ldots (\mathsf{div2}(b :: \alpha')_n \oplus w)) = \mathsf{half}(\alpha_0) \oplus (\mathsf{half}(\alpha_1) \oplus \ldots (\mathsf{half}(\alpha_n) \oplus z)),$$

where $\alpha' := \mathsf{tail}(\mathsf{tail}(\alpha))$. By using the inductive hypothesis on the sequence $b :: \alpha'$, this becomes

$$\langle a \rangle \oplus (\mathsf{half}(b) \oplus \ldots (\mathsf{half}(\alpha_n) \oplus z)) = \mathsf{half}(\alpha_0) \oplus (\mathsf{half}(\alpha_1) \oplus \ldots (\mathsf{half}(\alpha_n) \oplus z)).$$

Therefore, the result follows by Lemma 5.2.17.

We next show that halving an added sequence in the representation space realises the midpoint operation.

Lemma 5.2.19. [ϒ] *Given any ternary digits* $a, b : \mathbb{3}$, *we have that* $\mathsf{half}(\mathsf{add3}(a, b)) = \langle a \rangle \oplus \langle b \rangle$.

Proof. [f] By induction on $a, b : \mathbb{3}$, as well as the idempotency and commutativity of $\oplus$ by Definition 5.1.2.

Lemma 5.2.20. [ϒ] *Given any ternary signed-digit encodings* $\alpha, \beta : \mathbb{3}^{\mathbb{N}}$, *we have that* $M(\mathsf{map}(\mathsf{half}, \mathsf{zipWith}(\mathsf{add3}, \alpha, \beta))) = \langle\!\langle \alpha \rangle\!\rangle \oplus \langle\!\langle \beta \rangle\!\rangle$.

Proof. [f] By function extensionality and Lemma 5.2.19, $M(\mathsf{map}(\mathsf{half}, \mathsf{zipWith}(\mathsf{add}\mathbb{3}, \alpha, \beta))) = M(\lambda n.\langle\alpha\rangle \oplus \langle\beta\rangle)$. The result then follows by Lemma 5.1.12.

Using the two lemmas we have proved, we can now complete our verification of the midpoint operation.

Theorem 5.2.21. [ ] *The midpoint function on signed-digit encodings realises the midpoint operator on the interval object.*

Proof. [f] By Lemmas 5.2.18 and 5.2.20.

Infinitary midpoint

The infinitary midpoint function is obviously more complicated, both to define and verify. Escardó defines the infinitary midpoint on signed-digit encodings by using another intermediary signed-digit representation (this time of the interval $[-4, 4]$).

Definition 5.2.22. [ ] The type of *nonary digits* $\mathbb{9}$, equivalent to $\mathsf{Fin}(9)$, is defined by its elements $\overline{4}, \overline{3}, \overline{2}, \overline{1}, 0, 1, 2, 3, 4 : \mathbb{9}$.

Definition 5.2.23. [ ] We define the addition function on quinary digits $\mathsf{add}\mathbb{5} : \mathbb{5} \to \mathbb{5} \to \mathbb{9}$ in the expected way by pattern matching.

We also define the quartering function $\mathsf{div4} : \mathbb{9}^{\mathbb{N}} \to \mathbb{3}^{\mathbb{N}}$ by induction, and the related function $\mathsf{quarter} : \mathbb{9} \to \mathbb{I}$ on nonary digits.

Definition 5.2.24. [ ] We define the quartering function $\mathsf{div4} : \mathbb{9}^{\mathbb{N}} \to \mathbb{3}^{\mathbb{N}}$ from nonary signed-digit encodings to ternary signed-digit encodings similarly to how we defined $\mathsf{div2} : \mathbb{5}^{\mathbb{N}} \to \mathbb{3}^{\mathbb{N}}$.

Definition 5.2.25. [ ] We define the quartering map $\mathsf{quarter} : \mathbb{9} \to \mathbb{I}$ from nonary digits to the interval object similarly to how we defined $\mathsf{half} : \mathbb{5} \to \mathbb{I}$.

Lemma 5.2.26. [ ] *Given any ternary signed-digit encoding $\alpha : \mathbb{3}^{\mathbb{N}}$, we have that* $\langle\!\langle \mathsf{div4}(\alpha) \rangle\!\rangle = M(\mathsf{map}(\mathsf{quarter}, \alpha))$.

Proof. [f] Similar to Lemma 5.2.18.

We now define Adam Scriven's infinitary midpoint function on ternary signed-digit encodings in our framework. This definition was first given by Scriven in his MSc thesis, and reimplemented by Escardóin his Haskell library [Scr07; Esc11b].

Definition 5.2.27. [ ] The infinitary midpoint function on ternary signed-digit encodings is defined by translating Scriven's definition to Agda:

$$\begin{aligned}
&\mathsf{bigMid}' : (\mathbf{3}^{\mathbb{N}})^{\mathbb{N}} \to \mathbf{9}^{\mathbb{N}},\\
&\mathsf{bigMid}'(((a :: b :: x) :: (c :: y) :: \zeta))_0 &&:= \mathsf{add}\mathbf{5}(\mathsf{add}\mathbf{3}(a,a), \mathsf{add}\mathbf{3}(b,c)),\\
&\mathsf{bigMid}'(((a :: b :: x) :: (c :: y) :: \zeta))_{n+1} &&:= \mathsf{bigMid}'(\mathsf{mid}(x,y) :: \zeta)_n,
\end{aligned}$$

$$\begin{aligned}
&\mathsf{bigMid} : (\mathbf{3}^{\mathbb{N}})^{\mathbb{N}} \to K,\\
&\mathsf{bigMid} &&:= \mathsf{div4} \circ \mathsf{bigMid}'.
\end{aligned}$$

$$\begin{tikzcd}
(\mathbf{3}^{\mathbb{N}})^{\mathbb{N}} \arrow[r, "\mathsf{bigMid}'"] \arrow[rr, bend left, "\mathsf{bigMid}"] \arrow[d, "\mathsf{map}\ \langle - \rangle"'] & \mathbf{9}^{\mathbb{N}} \arrow[r, "\mathsf{div4}"] \arrow[d, "M \circ \mathsf{map}(\mathsf{quarter})"] & \mathbf{3}^{\mathbb{N}} \arrow[d, "\langle - \rangle"] \\
\mathbb{I}^{\mathbb{N}} \arrow[r, "M"'] & \mathbb{I} \arrow[r, equal] & \mathbb{I}
\end{tikzcd}$$

Figure 5.8: Commutative diagram illustrating Theorem 5.2.32.

This operation does indeed realise the iteration operator $M\colon \mathbb{I}^{\mathbb{N}} \to \mathbb{I}$ on the interval object; i.e. the diagram in Figure 5.8 commutes. We prove this by using the following lemmas.

Lemma 5.2.28. [ ] *Given any ternary digits $a, b, c\colon \mathbf{3}$, we have that* $\mathsf{quarter}(\mathsf{add}\mathbf{5}(\mathsf{add}\mathbf{3}(a,a), \mathsf{add}\mathbf{3}(b,c))) = \langle a \rangle \oplus (\langle b \rangle \oplus \langle c \rangle)$.

Proof. By induction on $a, b, c\colon \mathbf{3}$, as well as the idempotency, commutativity and transpositionality of $\oplus$ by Definition 5.1.2.

Lemma 5.2.29. [ ] *Given any ternary signed-digit encodings $\alpha, \beta\colon \mathbf{3}^{\mathbb{N}}$ and real* *number $z\colon \mathbb{I}$ in the interval object, we have,*

$$\langle\!\langle \alpha \rangle\!\rangle \oplus (\langle\!\langle \beta \rangle\!\rangle \oplus z) = (\langle a \rangle \oplus (\langle b \rangle \oplus \langle c \rangle)) \oplus (\langle\!\langle \mathsf{mid}(\alpha', \beta') \rangle\!\rangle \oplus z),$$

where $(a :: b :: \alpha') := \alpha$ and $(c :: \beta') := \beta$.

Proof. [f] By the following equational reasoning, (i) idempotency of M (Lemma 5.1.11), (ii) commutativity and (iii) transpositionality of $\oplus$ (Definition 5.1.2), and (iv) the correctness of the binary midpoint operation (Theorem 5.2.21),

$$\begin{array}{lll}
\langle\!\langle \alpha \rangle\!\rangle & \oplus (\langle\!\langle \beta \rangle\!\rangle \oplus z), & \\
:= \langle\!\langle (a :: (b :: \alpha')) \rangle\!\rangle & \oplus (\langle\!\langle (c :: \beta') \rangle\!\rangle \oplus z), & \\
= \langle a \rangle \oplus \langle\!\langle (b :: \alpha') \rangle\!\rangle & \oplus (\langle\!\langle (c :: \beta') \rangle\!\rangle \oplus z), & \text{by (i)}, \\
= (\langle a \rangle \oplus (\langle b \rangle \oplus \langle\!\langle \alpha' \rangle\!\rangle)) & \oplus (\langle\!\langle (c :: \beta') \rangle\!\rangle \oplus z), & \text{by (i)}, \\
= (\langle a \rangle \oplus (\langle b \rangle \oplus \langle\!\langle \alpha' \rangle\!\rangle)) & \oplus ((\langle c \rangle \oplus \langle\!\langle \beta' \rangle\!\rangle) \oplus z), & \text{by (i)}, \\
= ((\langle b \rangle \oplus \langle\!\langle \alpha' \rangle\!\rangle) \oplus \langle a \rangle) & \oplus ((\langle c \rangle \oplus \langle\!\langle \beta' \rangle\!\rangle) \oplus z), & \text{by (ii)}, \\
= ((\langle b \rangle \oplus \langle\!\langle \alpha' \rangle\!\rangle) \oplus (\langle c \rangle \oplus \langle\!\langle \beta' \rangle\!\rangle)) & \oplus (\langle a \rangle \oplus z), & \text{by (iii)}, \\
= ((\langle b \rangle \oplus \langle c \rangle) \oplus (\langle\!\langle \alpha' \rangle\!\rangle \oplus \langle\!\langle \beta' \rangle\!\rangle)) & \oplus (\langle a \rangle \oplus z), & \text{by (iii)}, \\
= ((\langle b \rangle \oplus \langle c \rangle) \oplus (\langle a \rangle)) & \oplus ((\langle\!\langle \alpha' \rangle\!\rangle \oplus \langle\!\langle \beta' \rangle\!\rangle) \oplus z), & \text{by (iii)}, \\
= (\langle a \rangle \oplus (\langle b \rangle \oplus \langle c \rangle)) & \oplus ((\langle\!\langle \alpha' \rangle\!\rangle \oplus \langle\!\langle \beta' \rangle\!\rangle) \oplus z), & \text{by (iii)}, \\
= (\langle a \rangle \oplus (\langle b \rangle \oplus \langle c \rangle)) & \oplus (\mathsf{mid}(\alpha', \beta') \oplus z)), & \text{by (iv)}.
\end{array}$$

Corollary 5.2.30. [✓] *Given any ternary signed-digit encodings $\alpha, \beta\colon \mathbb{3}^{\mathbb{N}}$ and real number $z\colon \mathbb{I}$ in the interval object, we have,*

$$\langle\!\langle \alpha \rangle\!\rangle \oplus (\langle\!\langle \beta \rangle\!\rangle \oplus z) = \mathsf{quarter}(\mathsf{add}\mathbb{5}(\mathsf{add}\mathbb{3}(a, a), \mathsf{add}\mathbb{3}(b, c))) \oplus (\langle\!\langle \mathsf{mid}(\alpha', \beta') \rangle\!\rangle \oplus z),$$

where $(a :: b :: \alpha') := \alpha$ and $(c :: \beta') := \beta$.

Proof. [f] By Lemmas 5.2.28 and 5.2.29.

Thus, it turns out that the bigMid operation provides an n-approximation of M for every $n\colon \mathbb{N}$.

Lemma 5.2.31. [✓] *The functions* $\mathsf{map}(\langle\!\langle - \rangle\!\rangle)$ *and* $\mathsf{map}(\mathsf{quarter}) \circ \mathsf{bigMid}'$ *— both of type $(\mathbb{3}^{\mathbb{N}})^{\mathbb{N}} \to \mathbb{I}^{\mathbb{N}}$ — are n-approximately equal for all $n\colon \mathbb{N}$.*

Proof. [f] By induction and Corollary 5.2.30.

Theorem 5.2.32. [ ] *The infinitary midpoint function on ternary signed-digit encodings realises the iteration operator on the interval object; i.e. given any sequence of ternary signed-digit encodings $\zeta : (\mathfrak{3}^{\mathbb{N}})^{\mathbb{N}}$, we have $\langle\!\langle \mathsf{bigMid}(\zeta) \rangle\!\rangle = M(\mathsf{map}(\langle\!\langle - \rangle\!\rangle, \zeta))$.*

Proof. [f] By Theorem 5.1.19 and Lemmas 5.2.26 and 5.2.31.

Multiplication

Finally, we define and verify multiplication. In the Haskell library, Escardó defined a variety of multiplication functions on signed-digit encodings; we choose to define and verify `mul_version0`, which uses the bigMid function [Esc11b].

Definition 5.2.33. [ ] The auxiliary multiplication function, which multiplies a ternary signed-digit encoding by a ternary digit, is defined as follows:

$$\begin{aligned}
&\mathsf{digitMul} : \mathfrak{3} \to \mathfrak{3}^{\mathbb{N}} \to \mathfrak{3}^{\mathbb{N}},\\
&\mathsf{digitMul}(\bar{1}, \beta) \quad := \mathsf{neg}(\beta),\\
&\mathsf{digitMul}(0, \beta) \quad := \lambda n.0,\\
&\mathsf{digitMul}(1, \beta) \quad := \beta.
\end{aligned}$$

Definition 5.2.34. [ ] The multiplication function on ternary signed-digit encodings is defined by multiplying the first argument against each individual digit of the second, and then taking the infinitary midpoint:

$$\begin{aligned}
&\mathsf{mul} : \mathfrak{3}^{\mathbb{N}} \to \mathfrak{3}^{\mathbb{N}} \to \mathfrak{3}^{\mathbb{N}},\\
&\mathsf{mul}(\alpha, \beta) \quad := \mathsf{bigMid}(\mathsf{zipWith}(\mathsf{digitMul}, \alpha, \mathsf{repeat}\ \beta)).
\end{aligned}$$

This function realises the multiplication function on the interval object; i.e. the diagrams in Figure 5.9 commute.

Lemma 5.2.35. [ ] *The auxiliary multiplication operator realises multiplication on the interval object; i.e. given any ternary digit $t : \mathfrak{3}$ and a ternary signed-digit encodings $\beta : \mathfrak{3}^{\mathbb{N}}$, we have $\langle\!\langle \mathsf{digitMul}(t, \beta) \rangle\!\rangle = \langle t \rangle * \langle\!\langle \beta \rangle\!\rangle$.*

$$\begin{tikzcd}
\mathfrak{3} \times \mathfrak{3}^{\mathbb{N}} \arrow[r, "\mathsf{digitMul}"] \arrow[d, "\langle - \rangle \times \langle - \rangle"'] & K \arrow[d, "\langle - \rangle"] \\
\mathbb{I} \times \mathbb{I} \arrow[r, "*"'] & \mathbb{I}
\end{tikzcd}
\qquad
\begin{tikzcd}
\mathfrak{3}^{\mathbb{N}} \times \mathfrak{3}^{\mathbb{N}} \arrow[r, "\mathsf{digitMul}'"] \arrow[rr, bend left, "\mathsf{mul}"] \arrow[d, "\langle - \rangle \times \langle - \rangle"'] & (\mathfrak{3}^{\mathbb{N}})^{\mathbb{N}} \arrow[r, "\mathsf{bigMid}"] \arrow[d, "\mathsf{map}\ \langle - \rangle"] & \mathfrak{3}^{\mathbb{N}} \arrow[d, "\langle - \rangle"] \\
\mathbb{I} \times \mathbb{I} \arrow[r, "\mathsf{repeat} \circ *"'] \arrow[rr, bend right, "*"'] & \mathbb{I}^{\mathbb{N}} \arrow[r, "M"'] & \mathbb{I}
\end{tikzcd}$$

Figure 5.9: Commutative diagrams illustrating Lemma 5.2.35 (left) and Theorem 5.2.36 (right). In the diagram, $\mathsf{digitMul}' := \lambda\alpha\beta.\mathsf{zipWith}(\mathsf{digitMul}, \alpha, \mathsf{repeat}\ \beta)$.

Proof. [f] By induction on $t : \mathfrak{3}$.

In the case where $t := \overline{1}$, we must show that $\langle\!\langle \mathsf{neg}(\beta) \rangle\!\rangle = -1 * \langle\!\langle \beta \rangle\!\rangle$. By the negation realiser (Theorem 5.2.10), $\langle\!\langle \mathsf{neg}(\beta) \rangle\!\rangle = -\langle\!\langle \beta \rangle\!\rangle$. The result follows by Lemma 5.1.41.

In the case where $t := 0$, we must show that $\langle\!\langle \lambda n.0 \rangle\!\rangle = 0 * \langle\!\langle \beta \rangle\!\rangle$. By the idempotency of M (Lemma 5.1.11), $\langle\!\langle \lambda n.0 \rangle\!\rangle := M(\mathsf{map}(\langle - \rangle, \lambda n.0)) = \langle 0 \rangle = 0$. The result follows by Lemma 5.1.43.

In the case where $t := 1$, we must show that $\langle\!\langle \beta \rangle\!\rangle = 1 * \langle\!\langle \beta \rangle\!\rangle$. The result follows by Lemma 5.1.42.

Theorem 5.2.36. [⊬] *The multiplication function on ternary signed-digit encodings realises the multiplication operation on the interval object.*

Proof. [f] By the following equational reasoning, (i) the correctness of the infinitary midpoint operation (Theorem 5.2.32), (ii) the correctness of the auxiliary multiplication operation (Lemma 5.2.35), (iii) the fact right-multiplication — i.e. $\lambda y.x * y$ for a given $x : \mathbb{I}$ — is a midpoint homomorphism (Lemma 5.1.45), and (iv) the fact that midpoint homomorphisms are M-homomorphisms (Lemma 5.1.15),

$$\begin{aligned}
& \langle\!\langle \mathsf{mul}(\alpha, \beta) \rangle\!\rangle, \\
:=& \langle\!\langle \mathsf{bigMid}(\mathsf{zipWith}(\mathsf{digitMul}, \alpha, \mathsf{repeat}\ \beta)) \rangle\!\rangle, \\
=& M(\mathsf{map}(\langle\!\langle - \rangle\!\rangle, \mathsf{zipWith}(\mathsf{digitMul}, \alpha, \mathsf{repeat}\ \beta))) && \text{by (i)}, \\
:=& M(\lambda n.\langle\!\langle \mathsf{digitMul}(\alpha_n, \beta) \rangle\!\rangle), \\
=& M(\lambda n.\langle \alpha_n \rangle * \langle\!\langle \beta \rangle\!\rangle) && \text{by (ii)}, \\
=& M(\lambda n.\langle \alpha_n \rangle) * \langle\!\langle \beta \rangle\!\rangle && \text{by (iii) and (iv)}, \\
:=& \langle\!\langle \alpha \rangle\!\rangle * \langle\!\langle \beta \rangle\!\rangle.
\end{aligned}$$

5.3 Ternary Boehm encodings

We shall see in Chapter 6 that the ternary signed-digit encodings are an ideal type for formalising the convergence of what we call ‘exact real search’. However, in that section it will also become clear that the efficiency of the algorithms extracted from the convergence theorems of search, optimisation and regression on that type leave much to be desired. For this reason, we introduce another type — ternary Boehm encodings — that yield more efficient algorithms at the expense of being more difficult to formalise in our AGDA framework[1].

In the 1990s, Hans-J. Boehm produced a practical JAVA library for exact real arithmetic [Boe99]. Since then, Boehm’s library has been developed to become the underpinning of Google’s bespoke Android calculator mobile phone application [Boe17; Boe20]. The star of the library is the class CR, objects of which Boehm calls *constructive reals*. Objects x of CR have a single private method `approximate`, which takes a (small) integer and outputs an (effectively unbounded) integer. The input integer n denotes the requested *precision-level* of the constructive real, while the output integer x_n denotes an integer approximation — scaled relative to the given precision-level — of the real number $[\![x]\!] : \mathbb{R}$ which x encodes.

Every object of CR is therefore a bi-infinite sequence of integer approximations of a real number, which is constructed and manipulated to ensure that the following condition about its relationship to the real number it encodes always holds:

$$d_{\mathbb{R}}([\![x]\!], 2^n x_n) < 2^n.$$

In this section, we will show how we re-rationalise this class — adapted slightly for the purposes of exact real search — within our AGDA framework as the type $\mathbb{T}$. We will then verify the structure of the type by defining the representation map $[\![-]\!] : \mathbb{T} \to \mathbb{R}$; for this purpose, we utilise the Dedekind reals rather than the interval object, both for the sake of variety and because elements of $\mathbb{T}$ represent reals without reference to a particular compact interval. However, following this, we will show how we use subtypes of $\mathbb{T}$ to represent only reals in particular compact intervals, as this is required for search. Finally, we will discuss how Boehm defines exact real arithmetic on CR by using interval arithmetic.

[1]Hence, we have not verified the ternary Boehm encodings to the same extent as the ternary signed-digit encodings.

5.3.1 Definition in our type theory

In order to define CR in our type theory, we start off by representing the class as functions $x\colon \mathbb{Z} \to \mathbb{Z}$. It is helpful to think of the integer approximation $x_n\colon \mathbb{Z}$, at any precision-level $n\colon \mathbb{Z}$, as an *interval approximation* of $[\![x]\!]$ — namely, x_n represents to the open interval $(2^n(x_n - 1), 2^n(x_n + 1))$, which clearly contains $[\![x]\!]$.

As we wish to use these encodings for search, the open interval structure is not ideal. Therefore, we alter CR by changing the order used in the relationship above from *strict* to *non-strict*. This is done because we require the use of compact intervals for search, though it makes little difference to the way CR objects are manipulated algorithmically. Furthermore, for stylistic reasons we make two further changes. Firstly, with the ternary signed-digits we used higher precision-levels n to mean *more* precise integer approximations, rather than less, and so we alter CR so that we can do the same. Furthermore, we slightly reposition the interval codes to avoid unnecessary subtraction in our formalisation.

By making the two changes above, the functions $x\colon \mathbb{Z} \to \mathbb{Z}$ we define for CR in our type theory must satisfy

$$d_{\mathbb{R}}([\![x]\!], 2^{-n}(x_n + 1)) \leq 2^{-n},$$

meaning that they are an infinitary sequence of integers, with each input-output pair $(x_n, n)\colon \mathbb{Z}^2$ representing the compact interval with dyadic rational endpoints

$$\left[\frac{x_n}{2^n}, \frac{x_n + 2}{2^n}\right].$$

We illustrate the structure of these encodings in Figure 5.10, which shows some possible interval approximations of real numbers in $[-3, 3]$ at precision-levels 0, 1 and 2 — note that the intervals halve in size as the precision-level increases further down the illustration. We refer to this as the *ternary structure*, because each interval represented by $(n, m)\colon \mathbb{Z}^2$ in the structure is perfectly trisected into three representations $(2n, m+1)$, $(2n + 1, m + 1)$ and $(2n + 2, m + 1)$ on the next precision-level.

This structure can be 'navigated' by the following operations on integer approximations. The operations downLeft, downMid and downRight take an integer that refers to an interval on an arbitrary precision-level and return one which refers, respectively, to one of the three possible trisections on the next precision-level. Meanwhile, upLeft and upRight do the reverse. The names here are colour coded with respect to the colours in

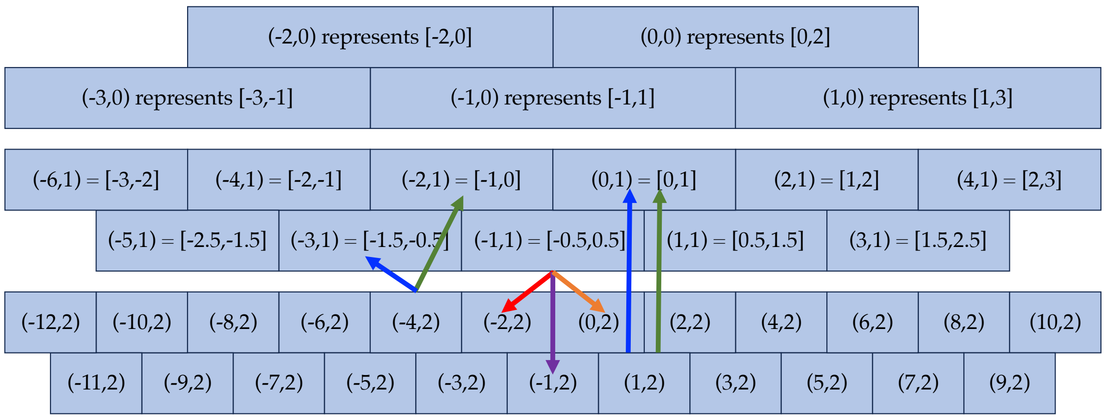


Figure 5.10: Ternary structure underlying the Boehm encodings. Each interval approximation is represented by an integer pair. The coloured arrows illustrate how the structural operations downLeft, downMid, downRight, upLeft and upRight work.

Figure 5.10, which illustrate how the operations work for particular intervals.

$$\begin{aligned}
\mathsf{downLeft}, \mathsf{downMid}, \mathsf{downRight}, \mathsf{upLeft}, \mathsf{upRight} &: \mathbb{Z} \to \mathbb{Z}, \\
\mathsf{downLeft}(k) &:= 2k, \\
\mathsf{downMid}(k) &:= 2k + 1, \\
\mathsf{downRight}(k) &:= 2k + 2, \\
\mathsf{upRight}(\mathsf{pos}\ k) &:= \mathsf{pos}\ (k/2), \\
\mathsf{upRight}(\mathsf{negsucc}\ k) &:= \mathsf{negsucc}\ (k/2), \\
\mathsf{upLeft}(k) &:= \mathsf{upRight}(k - 1).
\end{aligned}$$

Definition 5.3.1. [ ] [ ] Given two integers $n, m \colon \mathbb{Z}$, we say that n is *below* m if the interval represented by $(n, i+1) \colon \mathbb{Z}^2$ is a strict subinterval of that which is represented by $(m, i) \colon \mathbb{Z}^2$ for any $i \colon \mathbb{Z}$:

$$\begin{aligned}
\mathsf{below} &: \mathbb{Z} \to \mathbb{Z} \to \Omega, \\
n\ \mathsf{below}\ m &:= \mathsf{downLeft}\ m \leq n \leq \mathsf{downRight}\ m.
\end{aligned}$$

Equivalently, this can be defined as follows:

$$\begin{aligned}
\mathsf{below}' &: \mathbb{Z} \to \mathbb{Z} \to \Omega, \\
n\ \mathsf{below}'\ m &:= (n = \mathsf{downLeft}\ m) + (n = \mathsf{downMid}\ m) + (n = \mathsf{downRight}\ m).
\end{aligned}$$

To further illustrate how ternary Boehm encodings work, let us now give an example encoding $x\colon \mathbb{Z} \to \mathbb{Z}$ which is intended to represent the real number $2\colon \mathbb{R}$; x could be defined in a variety of ways:

- $x_0 = 1, x_1 = 2, \ldots$
- $x_0 = 1, x_1 = 3, \ldots$
- $x_0 = 1, x_1 = 4, \ldots$
- $x_0 = 0, x_1 = 2, \ldots$
- $x_0 = 0, x_1 = 3, \ldots$

All of these are valid representations of 2 as a ternary Boehm encoding because the two interval approximations *intersect*. However, we wish in our type theory to restrict ourselves to only those encodings that follow the ternary structure — the last item in the above list would not, therefore, be valid.

Definition 5.3.2. [🐓] A sequence of integers $x\colon \mathbb{Z} \to \mathbb{Z}$ is *ternary* if,

$$\mathsf{ternary}(x) := \prod_{(n\colon \mathbb{Z})} (x_{n+1} \ \mathsf{below}\ x_n)\,.$$

Restricting ourselves in this way will allow us to remove cases that are difficult to reason about, and to define continuity and search similarly to how we defined it for the ternary signed-digit encodings.

Definition 5.3.3. [🐓] The type of *ternary Boehm encodings* is defined as the collection of ternary sequences:

$$\mathbb{T} := \sum_{(x\colon \mathbb{Z}\to\mathbb{Z})} \mathsf{ternary}(x).$$

Returning to our illustration, we show one possible map that takes any integer to a ternary Boehm encoding that represents it.

Definition 5.3.4. The function that maps any integer to a ternary Boehm encodings is defined as follows:

$$\begin{aligned}
&\mathbb{Z}\text{-to-}\mathbb{T} : \mathbb{Z} \to \mathbb{T},\\
&\mathbb{Z}\text{-to-}\mathbb{T}(n)(\mathsf{pos}\ \ 0) &&:= n,\\
&\mathbb{Z}\text{-to-}\mathbb{T}(n)(\mathsf{pos}\ \ (i+1)) &&:= \mathsf{downLeft}^{i+1}(n),\\
&\mathbb{Z}\text{-to-}\mathbb{T}(n)(\mathsf{negsucc}\ \ i) &&:= \mathsf{upRight}^{i+1}(n).
\end{aligned}$$

Before proceeding, we invite the reader to verify in their minds that this map is indeed correct.

5.3.2 Verification of Boehm encodings via Dedekind reals

In this subsection, we will formally verify that the above definition of ternary Boehm encodings does indeed represent real numbers. The application of this work is for the future verification of exact real arithmetic on the Boehm encodings (discussed in Sections 5.3.3 and 7.2.2).

We first quickly recap the Dedekind reals, which we use for the type of real numbers $\mathbb{R}$ in this section. The work of implementing and formalising the Dedekind reals (as well as the rationals and dyadic rationals) for the AGDA library TYPETOPOLOGY was performed by Andrew Sneap in his M.Sci. project [Sne21].

The Dedekind reals can be defined over any dense subset of the real numbers, and are usually defined over the rationals $\mathbb{Q}$. We choose to instead define them over the *dyadic rationals* $\mathbb{Z}[1/2]$ — i.e. those rational numbers of form $\frac{z}{2^n}$ for $z, n : \mathbb{Z}$ — as this will be more convenient when verifying the structure of $\mathbb{T}$, whose interval approximations have dyadic endpoints. We use the dyadics informally in this thesis; simply noting they are a discrete set with a partial order, and that there is a function $\iota : \mathbb{Z} \times \mathbb{Z} \to \mathbb{Z}[1/2]$ which map pairs of integers $(z, n) : \mathbb{Z} \times \mathbb{Z}$ to the dyadic rational $\frac{z}{2^n}$.

Definition 5.3.5. [◎] A *Dedekind real number* $(L, R) : \mathbb{R}$ consists of two predicates on dyadics $L, R : \mathbb{Z}[1/2] \to \Omega$ (the *left cut* and the *right cut* respectively) that satisfy,

1. $\exists_{(p : \mathbb{Z}[1/2])} L(p)$,
2. $\exists_{(q : \mathbb{Z}[1/2])} R(p)$,
3. $\prod_{(p : \mathbb{Z}[1/2])} \left(L(p) \Leftrightarrow \exists_{(p' : \mathbb{Z}[1/2])} \left(L(p') \times p < p'\right)\right)$,
4. $\prod_{(q : \mathbb{Z}[1/2])} \left(R(q) \Leftrightarrow \exists_{(q' : \mathbb{Z}[1/2])} \left(R(q') \times q' < q\right)\right)$,
5. $\prod_{(p,q : \mathbb{Z}[1/2])} (L(p) \times R(q) \to p < q)$,
6. $\prod_{(p,q : \mathbb{Z}[1/2])} (p < q \to L(p) \vee R(q))$.

Now recall the definitions of representation maps in Section 5.2.2. In order to define the representation map $[\![-]\!] : \mathbb{T} \to \mathbb{R}$ on ternary Boehm encodings, we first look at how one can represent real numbers as sequences of dyadic rational intervals that satisfy the *nested* and *positioned* properties (defined below). We will then show that — as we illustrated in the previous subsection — every ternary Boehm real $\chi : \mathbb{T}$ can be transformed into such a sequence of dyadic intervals, and therefore that we can indeed define $[\![\chi]\!] : \mathbb{R}$.

Definition 5.3.6. [ ] The type of *valid representations of dyadic intervals* $\mathbb{Z}[1/2]^I$ is defined as the type of pairs of dyadics such that the first is smaller than the second:

$$\mathbb{Z}[1/2]^I :=\sum_{((l,r)\colon \mathbb{Z}[1/2]\times\mathbb{Z}[1/2])} (l \leq r)\,.$$

For convenience, we simply refer to elements of this type as *dyadic intervals*.

Definition 5.3.7. [ ] One dyadic interval *covers* another if,

$$\begin{aligned} \mathsf{covers}\ &: \mathbb{Z}[1/2]^I \longrightarrow \mathbb{Z}[1/2]^I \longrightarrow \Omega, \\ (l_1, r_1)\ \mathsf{covers}\ (l_2, r_2) &:= (l_1 \leq l_2) \times (r_2 \leq r_1). \end{aligned}$$

Definition 5.3.8. [ ] A sequence of dyadic intervals $\chi\colon \mathbb{Z} \longrightarrow \mathbb{Z}[1/2]^I$ is *nested* if every interval in the sequence covers the next:

$$\begin{aligned} &\mathsf{nested} : (\mathbb{Z} \longrightarrow \mathbb{Z}[1/2]^I) \longrightarrow \Omega, \\ &\mathsf{nested}(\chi) := \Pi_{(n\colon \mathbb{N})}\,(\chi_n\ \mathsf{covers}\ \chi_{n+1})\,. \end{aligned}$$

Definition 5.3.9. [ ] A sequence of dyadic intervals $\chi\colon \mathbb{Z} \longrightarrow \mathbb{Z}[1/2]^I$ is *positioned* if it contains arbitrary small intervals,

$$\begin{aligned} &\mathsf{positioned} : (\mathbb{Z} \longrightarrow \mathbb{Z}[1/2]^I) \longrightarrow \Omega, \\ &\mathsf{positioned}(\chi) := \Pi_{(\varepsilon\colon \mathbb{Z}[1/2])}(\varepsilon > 0) \longrightarrow \Sigma_{(n\colon \mathbb{N})}\,(r_n - l_n \leq \varepsilon) \quad \textit{(where } (l_n, r_n) := \chi_n). \end{aligned}$$

Lemma 5.3.10. [ ] [a] *If a sequence of dyadic intervals $\chi\colon \mathbb{Z} \longrightarrow \mathbb{Z}[1/2]^I$ is nested and positioned, there is a real number $(\!|\chi|\!)\colon \mathbb{R}$ which it represents.*

[a]In Agda, the formalisation of this proof currently requires the use of a number of unformalised lemmas about dyadic numbers.

Proof. [t] The proof is due to Andrew Sneap. The Dedekind cuts $L, R\colon \mathbb{Z}[1/2] \longrightarrow \Omega$ are defined as follows:

$$\begin{aligned} L(p) &:= \exists_{(n\colon \mathbb{Z})}\,(p < \mathsf{pr}_1(\chi_n))\,, \\ R(q) &:= \exists_{(n\colon \mathbb{Z})}\,(\mathsf{pr}_2(\chi_n) < q)\,. \end{aligned}$$

The idea of L is that a rational number is less than the represented number if there is an interval in the sequence that it is smaller than the lower endpoint of. The idea of R is the opposite of this. The first four properties of the cuts (Definition 5.3.5) are proved by properties of the dyadics, while the fifth is by the nested property and the sixth is by the positioned property.

Any interval that has dyadic endpoints with the same denominator can be represented as a triple of integers.

Definition 5.3.11. [✓] We call a triple of integers $(k, c, p) \colon \mathbb{Z}^3$ such that $k \leq c$ (the proof term of which we leave implicit) a *dyadic interval code* when we use it for the purpose of representing the dyadic interval $\mathbb{Z}^3\text{-to-}\mathbb{Z}[1/2]^{\mathrm{I}}(k, c, p) := (\iota(k, p), \iota(c, p)) \colon \mathbb{Z}[1/2]^I$ (which has width $\frac{c-k}{2^p}$).

Given a sequence of dyadic interval codes $\chi \colon \mathbb{Z} \to \mathbb{Z}^3$, we overload terminology by saying χ is nested/positioned to mean that $\mathsf{map}(\mathbb{Z}^3\text{-to-}\mathbb{Z}[1/2]^{\mathrm{I}}, \chi)$ is nested/positioned.

Corollary 5.3.12. [✓] *If a sequence of dyadic interval codes $\chi \colon \mathbb{Z} \to \mathbb{Z}^3$ is nested and positioned, it represents the real number $(\!|\chi|\!)' := (\!|\mathsf{map}(\mathbb{Z}^3\text{-to-}\mathbb{Z}[1/2]^{\mathrm{I}}, \chi)|\!) \colon \mathbb{R}$.*

Proof. [t] By Lemma 5.3.10.

The dyadic intervals underlying the structure of $\mathbb{T}$ (illustrated in Figure 5.10) can be represented by pairs of integers.

Definition 5.3.13. [✓] We call a pair of integers $(k, p) \colon \mathbb{Z}^2$ a *ternary interval code* when we use it for the purpose of representing the interval $\mathbb{Z}^2\text{-to-}\mathbb{Z}[1/2]^{\mathrm{I}}(k, p) := (\iota(k, p), \iota(k+2, p)) \colon \mathbb{Z}[1/2]^I$ (which has width $\frac{1}{2^{p-1}}$).

Furthermore, every ternary interval code (k, p) gives a dyadic interval code

$$\mathbb{Z}^2\text{-to-}\mathbb{Z}^3(k, p) := (k, k+2, p),$$

such that the below diagram commutes.

$$\begin{tikzcd}
\mathbb{Z}^2 \arrow[r, "\mathbb{Z}^2\text{-to-}\mathbb{Z}^3"] \arrow[dr, "\mathbb{Z}^2\text{-to-}\mathbb{Z}[1/2]^{\mathrm{I}}"'] & \mathbb{Z}^3 \arrow[d, "\mathbb{Z}^3\text{-to-}\mathbb{Z}[1/2]^{\mathrm{I}}"] \\
& \mathbb{Z}[1/2]^I
\end{tikzcd}$$

Given a sequence of ternary interval codes $\chi \colon \mathbb{Z} \to \mathbb{Z}^2$, we overload terminology by saying χ is nested/positioned to mean that $\mathsf{map}(\mathbb{Z}^2\text{-to-}\mathbb{Z}[1/2]^{\mathsf{I}}, \chi)$ is nested/positioned.

Corollary 5.3.14. [✓] *If sequence of ternary interval codes $\chi : \mathbb{Z} \to \mathbb{Z}^2$ is nested and positioned, it represents the real number $(\!|\chi|\!)'' := (\!|\mathsf{map}(\mathbb{Z}^2\text{-to-}\mathbb{Z}^3, \chi)|\!)' : \mathbb{R}$.*

Proof. [t] By Corollary 5.3.12.

For sequences of ternary interval codes, being positioned is implied by a stronger property we call the *normalised* property. A sequence of such interval codes is normalised if the precision-level of the output interval is always identical to that which was requested.

Definition 5.3.15. [✓] A sequence of ternary interval codes is *normalised* if the precision-level of the code at every point $n : \mathbb{N}$ is always exactly n.

$$\begin{aligned}&\mathsf{normalised} : (\mathbb{Z} \to \mathbb{Z}^2) \to \Omega,\\ &\mathsf{normalised}(\chi) := \Pi_{(n:\,\mathbb{N})}\left(\mathsf{pr}_2(\chi_n) = n\right).\end{aligned}$$

Lemma 5.3.16. [✓] [a] *If a sequence of ternary interval codes is normalised, then it is positioned.*

[a]In Agda, the formalisation of this proof currently requires the use of a number of unformalised lemmas about dyadic numbers.

Proof. Recall from Definition 5.3.9 that, given some $\varepsilon : \mathbb{Z}[1/2] > 0$ we wish to find an interval in the sequence whose width is at most ε.
We define $q : \mathbb{Z}$ to be any integer such that $\frac{2}{2^q} < \varepsilon$. By the normalised property, $q = pr_2(\chi_q)$ and, by definition of ternary interval codes, χ_q has width $\frac{2}{2^q}$, which is smaller than ε.

If a sequence of ternary interval codes $\chi : \mathbb{Z} \to \mathbb{Z}^2$ is normalised, we can effectively discard the precision-level information in the output and simply consider it as an integer sequence $\mathsf{map}(\mathsf{pr}_1, \chi) : \mathbb{Z} \to \mathbb{Z}$. Further, if χ is nested, then $\mathsf{map}(\mathsf{pr}_1, \chi)$ is ternary (Definition 5.3.2).

Lemma 5.3.17. [✓] [a] *Given a normalised sequence of ternary interval codes, it is nested if and only if it is ternary.*

[a]In Agda, the formalisation of the proof of this lemma currently requires the use of a number of unformalised lemmas about dyadic numbers.

As the resulting integer sequence is ternary, it is clearly a ternary Boehm encoding (Definition 5.3.3). Indeed, the type of ternary interval codes that are nested and positioned are *equivalent* to the type of ternary Boehm encodings $\mathbb{T}$.

Definition 5.3.18. [ᚠ] We define the inclusion map from ternary Boehm encodings to sequences of ternary interval codes by the following:

$$\begin{aligned}&\mathsf{to\text{-}interval\text{-}seq} : \mathbb{T} \to (\mathbb{Z} \to \mathbb{Z}^2),\\ &\mathsf{to\text{-}interval\text{-}seq}(\chi)_n := (\chi_n, n).\end{aligned}$$

Lemma 5.3.19. [ᚠ] *The type of nested and normalised sequences of ternary interval codes that are equivalent to the type of ternary Boehm encodings:*

$$\sum_{(\chi : \mathbb{Z} \to \mathbb{Z}^2)} (\mathsf{nested}(\mathsf{map}(\mathsf{pr}_1, \chi)) \times \mathsf{normalised}(\chi)) \simeq \mathbb{T}.$$

Proof. [f] Converting from $\mathbb{T} \to (\mathbb{Z} \to \mathbb{Z}^2)$ is done by to-interval-seq, while converting in the other direction is done by $\mathsf{map}(\mathsf{pr}_1)$; in both cases, we prove the necessary properties of the constructed representation using Lemma 5.3.17. It is then trivial to see that these two functions yield identities.

Corollary 5.3.20. [ᚠ] *Any given ternary Boehm encoding $x : \mathbb{T}$ represents the real number*

$$[\![x]\!] := (\!|\lambda n.(x_n, n)|\!)'' : \mathbb{R}.$$

Proof. [ft] By Lemma 5.3.19 and Corollary 5.3.14.

5.3.3 Exact real arithmetic

Boehm defines arithmetic operations on CR by looking to the interval arithmetic on the dyadic rational intervals [Boe20]. In this subsection we follow this same process in order to informally define negation, addition and multiplication on $\mathbb{T}$. The definitions here differ to those on CR because elements of $\mathbb{T}$ are different to those of CR (for example we must satisfy the ternary property, Definition 5.3.2); furthermore, we give simpler, more convenient definitions than Boehm's, though this is at the expense of some efficiency.

Note that the definitions of this subsection are informal — we do not define these operations in our formal library and have not verified their correctness; indeed, this is ongoing work as discussed in Section 7.2.2. We will, however, use these definitions in our own proof-of-concept Java library for exact real search on ternary Boehm encodings in Chapter 6.

Negation

Negation on dyadic intervals is defined by

$$-\left[\frac{k}{2^p}, \frac{c}{2^p}\right] := \left[\frac{-c}{2^p}, \frac{-k}{2^p}\right].$$

Hence, negation on dyadic interval codes $\mathbb{Z}^3$ is defined by $-(k, c, p) := (-c, -k, p)$. The width of the output interval is the same as that of the input interval, meaning that if the input was in fact a ternary interval code $\mathbb{Z}^2$, then the output also would be a ternary interval code on the same precision-level. Therefore, we can easily define negation on ternary Boehm encodings.

Definition 5.3.21. *Negation* on ternary Boehm encodings is defined by the following:

$$\begin{aligned} - &: \mathbb{T} \to \mathbb{T}, \\ -x &:= \lambda n. - x_n - 2. \end{aligned}$$

Addition

Addition on dyadic intervals is defined by

$$\left[\frac{k_1}{2^{p_1}}, \frac{c_1}{2^{p_1}}\right] + \left[\frac{k_2}{2^{p_2}}, \frac{c_2}{2^{p_2}}\right] := \left[\frac{2^{p_2-\min(p_1,p_2)}k_1 + 2^{p_1-\min(p_1,p_2)}k_2}{2^{\max(p_1,p_2)}}, \frac{2^{p_2-\min(p_1,p_2)}c_1 + 2^{p_1-\min(p_1,p_2)}c_2}{2^{\max(p_1,p_2)}}\right].$$

As we are looking to define this operation on $\mathbb{T}$, whereon we can approximate the elements to any precision-level we like, we can consider the simpler case where the precision-levels of the inputs are identical. Hence, addition on dyadic interval codes $\mathbb{Z}^3$ in the case where the precision-levels of the inputs is identical is defined by $(k_1, c_1, p) + (k_2, c_2, p) := (k_1 + k_2, c_1 + c_2, p)$. The width of the output interval in this case is

$$\frac{(c_1 + c_2) - (k_1 + k_2)}{2^p},$$

meaning that given two ternary interval codes on precision-level p the width would be

$$\frac{(k_1 + 2 + k_2 + 2) - (k_1 + k_2)}{2^p} = \frac{1}{2^{p-2}}.$$

In order to achieve an output interval approximation of a ternary Boehm encoding at a requested precision-level $p\colon \mathbb{Z}$, therefore, the input ternary Boehm encodings must be evaluated to precision-level $p + 2$.

Definition 5.3.22. *Addition* on ternary Boehm encodings is defined by the following:

$$\begin{aligned} + \quad &: \mathbb{T} \to \mathbb{T} \to \mathbb{T}, \\ x + y &:= \lambda n.(x_{n+2} + y_{n+2}). \end{aligned}$$

Multiplication

Multiplication on dyadic intervals is defined by

$$\left[\frac{k_1}{2^{p_1}}, \frac{c_1}{2^{p_1}}\right] * \left[\frac{k_2}{2^{p_2}}, \frac{c_2}{2^{p_2}}\right] := \left[\frac{\mathsf{min}(k_1k_2, k_1c_2, c_1k_2, k_2c_2)}{2^{p_1+p_2}}, \frac{\mathsf{max}(k_1k_2, k_1c_2, c_1k_2, k_2c_2)}{2^{p_1+p_2}}\right].$$

As with addition, we can consider the simpler case where the precision-levels of the inputs are identical. Hence, multiplication on dyadic interval codes $\mathbb{Z}^3$ in the case where the precision-levels of the inputs is identical is defined by $(k_1, c_1, p) * (k_2, c_2, p) := (\mathsf{min}(k_1k_2, k_1c_2, c_1k_2, k_2c_2), \mathsf{max}(k_1k_2, k_1c_2, c_1k_2, k_2c_2), 2p)$. The width of the output interval varies based on the inputs — this is related to the fact that multiplication is continuous but not uniformly continuous. In the case where both intervals are positive, for example, the width of the output interval is

$$\frac{((k_1 + 2)(k_2 + 2)) - k_1k_2}{2^{2p}} = \frac{k1 + k2 + 2}{2^{2p-1}} = \frac{1}{2^{2p-1-\mathsf{log2}(k1+k2+2)}}.$$

This case in fact supercedes all others, and so in order to achieve an output interval approximation of a ternary Boehm encoding at a requested precision-level $p\colon \mathbb{Z}$, the input ternary Boehm encodings must be evaluated to precision-level $(p + \mathsf{log2}(\mathsf{abs}(k_1) + \mathsf{abs}(k_2) + 2) + 1)/2$.

Definition 5.3.23. *Multiplication* on ternary Boehm encodings is defined by the following:

$$\begin{aligned} * \quad &: \mathbb{T} \to \mathbb{T} \to \mathbb{T}, \\ x * y &:= \lambda n.(x_{(n+\text{log2}(\text{abs}(x_n)+\text{abs}(y_n)+2)+1)/2} + y_{(n+\text{log2}(\text{abs}(x_n)+\text{abs}(y_n)+2)+1)/2}). \end{aligned}$$

5.3.4 Representing compact intervals

A key difference between ternary Boehm encodings $\mathbb{T}$ and ternary signed-digit encodings $\mathbf{3}^{\mathbb{N}}$ is that with the former we can represent real numbers across the real line, whereas with the latter we can only represent reals in a given compact interval (in this thesis, we use the example of $[-1, 1]$). However, $\mathbb{T}$ is not a uniformly continuous searchable type — in order to search the ternary Boehm encodings, we will have to restrict ourselves to particular compact intervals. In this subsection, we define three subtypes of $\mathbb{T}$ that can be used to represent real numbers that are in the compact interval $[\frac{k}{2^i}, \frac{k+2}{2^i}]$, which is represented by the ternary interval code $(k, i) : \mathbb{Z}^2$.

The first subtype $\mathbb{T}(k, i)_1$ is straightforward to define: it is simply the type of ternary Boehm encodings which 'pass through' the interval encoded by (k, i) at precision-level i.

Definition 5.3.24. [ᚠ] For every ternary interval code $(k, i) : \mathbb{Z}^2$ there is a subtype $\mathbb{T}(k, i)_1$ of *ternary Boehm encodings in the compact interval* $[\frac{k}{2^i}, \frac{k+2}{2^i}]$ defined as the collection of ternary Boehm encodings that feature the interval approximation (k, i):

$$\mathbb{T}(k, i)_1 := \sum_{(x : \mathbb{T})} (x_i = k)\,.$$

This subtype is useful because it is easy to map back to elements of $\mathbb{T}$ by simply taking the first projection.

The second subtype $\mathbb{T}(k, i)_2$ is a little less convenient to work with directly, but is more convenient for the purpose of defining a closeness function (which we will do in Chapter 6). Elements $x : \mathbb{T}(k, i)_2$ are $\mathbb{N}$-indexed sequences that only give the interval approximations of the represented real number $[\![x]\!]$ for precision-levels greater than i — i.e. for any $n : \mathbb{N}$, an interval approximation of $[\![x]\!]$ is represented by the interval code $(x_n, i + 1 + n) : \mathbb{Z}^2$.

Definition 5.3.25. [ᚠ] For every ternary interval code $(k, i) : \mathbb{Z}^2$ there is a subtype $\mathbb{T}(k, i)_2$ of *ternary Boehm encodings in the compact interval* $[\frac{k}{2^i}, \frac{k+2}{2^i}]$ defined as the collection of sequences of type $\mathbb{Z}^{\mathbb{N}}$ that locate, after the interval approximation (k, i), any ternary Boehm encoding:

$$\mathbb{T}(k,i)_2 := \sum_{(\chi\colon \mathbb{Z}^{\mathbb{N}})} \left(\chi_0 \text{ below } k \times \Pi_{(n\colon \mathbb{N})} \left(\chi_{n+1} \text{ below } \chi_n\right)\right).$$

We can convert from each of these definitions to the other.

Definition 5.3.26. [ᚠ] Given any ternary interval code $(k,i)\colon \mathbb{Z}^2$, the function

$$\mathbb{T}(k,i)_1\text{-to-}\mathbb{T}(k,i)_2 : \mathbb{T}(k,i)_1 \to \mathbb{T}(k,i)_2$$

is defined for every $((x,b),e)\colon \mathbb{T}(k,i)_1$ where

- $x\colon \mathbb{Z} \to \mathbb{Z}$,
- $b\colon \mathsf{ternary}(x)$,
- $e\colon x_i = k$.

We first define the sequence

$$\chi : \mathbb{Z}^{\mathbb{N}},$$
$$\chi_n := x_{i+1+n}.$$

The proof that $(\chi_0 \text{ below } k)$ is then by $b_i\colon (x_{i+1} \text{ below } x_i)$ and e; the proof that $\Pi_{(n\colon \mathbb{N})} (\chi_{n+1} \text{ below } \chi_n)$ is by b.

Definition 5.3.27. [ᚠ] Given any ternary interval code $(k,i)\colon \mathbb{Z}^2$, the function

$$\mathbb{T}(k,i)_2\text{-to-}\mathbb{T}(k,i)_1 : \mathbb{T}(k,i)_2 \to \mathbb{T}(k,i)_1$$

is defined for every $(\chi, b_0, b_s)\colon \mathbb{T}(k,i)_2$ where

- $\chi\colon \mathbb{Z}^{\mathbb{N}}$,
- $b_0\colon \chi_0 \text{ below } k$,
- $b_s\colon \Pi_{(n\colon \mathbb{N})} (\chi_{n+1} \text{ below } \chi_n)$.

We first define the sequence

$$\begin{aligned} &x : \mathbb{Z} \to \mathbb{Z}, \\ &x_n := k && \textit{(when } n = i\textit{)}, \\ &x_n := \chi_{n-1-i} && \textit{(when } n > i\textit{)}, \\ &x_n := \mathsf{upRight}(k) && \textit{(when } n < i\textit{)}. \end{aligned}$$

The proof that $(\mathsf{ternary}(x))$ is then by b_0 and b_s; the proof that $x_n = k$ is immediate.

Despite the fact that, given a particular interval code $(k, i) \colon \mathbb{Z}^2$, the types $\mathbb{T}(k, i)_1$ and $\mathbb{T}(k, i)_2$ can represent the same real numbers, the types are not equivalent. This is because $\mathbb{T}(k, i)_1$ contains more information about locating the represented real number than $\mathbb{T}(k, i)_2$, and this information cannot be recovered when converting from the latter to the former — indeed, in Definition 5.3.27 we used the upRight function to prepend arbitrary interval approximations of the real number for precision-levels higher than i. Despite this, it *is* the case that the composition of the two functions defined above preserves the represented real number. The informal idea of this is that from precision-level $i \colon \mathbb{Z}$ the ternary Boehm encoding (and thus, the underlying dyadic interval approximations) are identical — hence, the limit of the two sequences is the same real number.

We now connect the two representations of real numbers in compact intervals that we use in this thesis, and show that the second subtype defined in this subsection is equivalent to the type of ternary Boehm encodings.

Lemma 5.3.28. [🐓] *Given any ternary interval code $(k, i) \colon \mathbb{Z}^2$, it is the case that $\mathbb{T}(k, i)_2 \simeq \mathbb{3}^{\mathbb{N}}$.*

Proof (Sketch). [f] Although the formalisation is rather involved, the intuition here is clear. We convert any ternary Boehm encoding in a compact interval $\chi \colon \mathbb{T}(k, i)_2$ into a ternary signed-digit encoding $\alpha \colon \mathbb{3}^{\mathbb{N}}$ by using the values of $b_0 \colon \chi_0$ below k and $b_s \colon \prod_{(n \colon \mathbb{N})} \chi_{n+1}$ below χ_n. If, at any point $n \colon \mathbb{N}$ in the sequence, χ_n goes downLeft then $\alpha_n = \overline{1}$; if χ_n goes downMid then $\alpha_n = 0$; and else if χ_n goes downRight then $\alpha_n = 1$. Following the opposite method, we can also convert in the other direction; and clearly either composition of these conversions gives an identity.

Finally, we give the third subtype for representing compact intervals using $\mathbb{T}$. We can remove the redundant interval approximations from the structure of $\mathbb{T}(k, i)_2$, which reflects the ability to represent the same real number as any ternary signed-digit encoding $\mathbb{3}^{\mathbb{N}}$ as a function $\mathbb{N} \to \mathbb{2}$ (discussed in Section 5.2.1).

Definition 5.3.29. [🐓] Given two integers $n, m \colon \mathbb{Z}$, we say that n is *split-below* m if

n is either downLeft m or downRight m:

$$\begin{aligned} \text{split-below} \quad &: \mathbb{Z} \to \mathbb{Z} \to \Omega, \\ n \text{ split-below } m &:= (n = \text{downLeft } m) + (n = \text{downRight } m). \end{aligned}$$

Definition 5.3.30. [ꟼ] For every ternary Boehm interval code $(k, i) : \mathbb{Z}^2$ there is a subtype $\mathbb{T}(k, i)_3$ of *ternary Boehm encodings in the compact interval* $[\frac{k}{2^i}, \frac{k+2}{2^i}]$ defined as the collection of sequences of type $\mathbb{Z}^{\mathbb{N}}$ that locate, after the interval approximation (k, i), only those ternary Boehm encodings that never use downMid:

$$\mathbb{T}(k, i)_3 := \sum_{(\chi : \mathbb{N} \to \mathbb{Z})} \left(\chi_0 \text{ split-below } k \times \Pi_{(n : \mathbb{N})} \left(\chi_{n+1} \text{ split-below } \chi_n\right)\right).$$

Therefore, this subtype defined is equivalent to the Cantor type $\mathbb{2}^{\mathbb{N}}$, as at each interval approximation there are two choices for the next.

Lemma 5.3.31. [ꟼ] *Given any ternary Boehm interval code* $(k, i) : \mathbb{Z}^2$*, it is the case that* $\mathbb{T}(k, i)_3 \simeq \mathbb{2}^{\mathbb{N}}$.

Proof. [f] Similar to Lemma 5.3.28.

Split-belowness trivially implies belowness, and hence every element of $\mathbb{T}(k, i)_3$ can be mapped to one in $\mathbb{T}(k, i)_2$ — and, hence, one in $\mathbb{T}(k, i)_3$ — all of which represent the same real number.

As discussed in Section 5.2.1, there is a problem when using the Cantor type $\mathbb{2}^{\mathbb{N}}$ for representing real numbers: namely, it is unsuitable for computation due to its lack of redundant digits. This problem extends to the equivalent subtype $\mathbb{T}(k, i)_3$, and so it is important to understand that we do not (and indeed cannot) use this type for computation in our framework. The reason for its introduction here is instead for the purposes of *search* — in Section 6.2.1, we will see that it is this subtype which is most suitable for searching the type of ternary Boehm encodings in the given compact interval $\left[\frac{k}{2^i}, \frac{k+2}{2^i}\right]$ (for any $(k, i) : \mathbb{Z}^2$). Indeed, this further means that the type $\mathbb{2}^{\mathbb{N}}$ is suitable for searching ternary signed-digit encodings, as we explore in Section 6.1.1.

Chapter 6

Exact Real Search

Our generalised framework for search, optimisation and regression introduced in Chapters 3 and 4 has been shown to operate on a wide class of types. Optimisation takes place on totally bounded (Definition 3.2.35) closeness spaces (Definition 3.2.19) equipped with a preorder (Definition 4.1.4) and approximate linear preorder (Definition 4.1.13); whereas search takes place on uniformly continuously searchable (Definition 3.3.3) closeness spaces — further, recall that totally boundedness implies uniformly continuous searchability (Theorem 3.3.6). Regression can take place on either of the above classes of closeness space, depending on whether we perform it via optimisation (Theorem 4.2.6) or search (Theorems 4.2.9 and 4.2.10).

In this thesis' final chapter, we bring this generalised framework full circle by instantiating it on the two representations of real numbers we explored and verified in Chapter 5: ternary signed-digits and ternary Boehm encodings. In order to achieve this, we formalise that each type's representations of real numbers in compact intervals — i.e. $\mathfrak{Z}^{\mathbb{N}}$ itself for the former and the subtype $\mathbb{T}(k,i)_3$ for any given $(k,i) : \mathbb{Z}^2$ (Definition 5.3.25) for the latter — are members of the aforementioned class of types, in that we can perform search, optimisation and regression on them. We then formalise, only for $\mathfrak{Z}^{\mathbb{N}}$, that the exact real arithmetic functions we wish to search, optimise and regress are indeed uniformly continuous functions — proofs that are required for the algorithms to run. Finally, we give a variety of toy, proof-of-concept examples of search, optimisation and regression using these functions.

On signed-digit encodings, our example Haskell algorithms are compiled directly from instantiations of the formal Agda proofs of convergence (we explain how in the final paragraph of Appendix A). This means that, while the extracted computation is relatively inefficient, we have formalised the fact that it will compute a correct answer. Meanwhile, our examples of the framework working on ternary Boehm encodings are written in a JAVA library which informally reflects the ideas of the formal framework; we sacrifice direct proofs of convergence in order to attain more efficient proof-of-concept algorithms.

6.1 Exact Real Search using signed-digit encodings

Some of the work of this section was previously published as part of a joint paper with Dan R. Ghica at the *Logic in Computer Science (LICS) 2021* conference [GA21].

6.1.1 Suitability for search, optimisation and regression

In this subsection, we show that the type of ternary signed digit encodings $\mathbf{3}^{\mathbb{N}}$ — explored in Section 5.2 — is a member of our class of types in that we can perform search, optimisation and regression upon it. We show, therefore, that $\mathbf{3}^{\mathbb{N}}$ is a totally bounded and uniformly continuously searchable closeness space with a preorder and approximate linear preorder.

$\mathbf{3}^{\mathbb{N}}$ is a continuously searchable closeness space

The closeness space on $\mathbf{3}^{\mathbb{N}}$ is a discrete-sequence closeness space (Definition 3.2.56), as the type $\mathbf{3}$ is finite and, therefore, discrete. Further, by the finiteness of $\mathbf{3}$, this closeness space is totally bounded.

Corollary 6.1.1. [ᚠ] *The type of ternary signed-digit encodings $\mathbf{3}^{\mathbb{N}}$ is a totally bounded closeness space.*

Proof. [f] Because $\mathbf{3}$ is finite and discrete, the result follows from Corollary 3.2.60.

There are two proofs of uniformly continuous searchability we can employ for $\mathbf{3}^{\mathbb{N}}$. The first is directly from the above fact that $\mathbf{3}^{\mathbb{N}}$ is totally bounded: this yields the *totally bounded uniformly continuous searcher.*

Corollary 6.1.2. [ᚠ] [ᚠ] *The closeness space of ternary signed-digit encodings $\mathbf{3}^{\mathbb{N}}$ is uniformly continuously searchable.*

Proof 1. By Corollary 6.1.1 and Theorem 3.3.6.

The second proof is inspired instead by Berger's searcher on the Cantor space and Escardó's on countable product spaces (which we formalised in Section 3.3.3): this yields the *decreasing-modulus uniformly continuous searcher.*

Proof 2. By Corollary 3.3.12.

In the examples given in Section 6.1.3, it will become apparent that the totally bounded searcher is almost always more efficient than the decreasing-modulus searcher; specifically, the decreasing-modulus searcher is only better in Example 6.1.35.

By combining the proofs of Corollary 6.1.2 with the examples in Sections 3.2.5, 3.3.2 and 3.3.3, we further have (among other things) that arbitrary products of $\mathbb{3}^{\mathbb{N}}$ are uniformly continuously searchable and totally bounded closeness spaces. However, we do not usually wish to search $\mathbb{3}^{\mathbb{N}}$ *directly*, because (as discussed in Section 5.2) there are infinitely-many redundant representations of any number that features a 0, and so a direct searcher will have inherent inefficiencies. Instead, we search $\mathbb{3}^{\mathbb{N}}$ *indirectly* using the Cantor type $\mathbb{2}^{\mathbb{N}}$. Recall that every real number in $[-1, 1]$ can be represented as a sequence $\mathbb{N} \to \{-1, 1\}$ and, hence, by using 0 for -1, as a sequence $\mathbb{2}^{\mathbb{N}}$. We can therefore trivially define the following inclusion map from $\mathbb{2}^{\mathbb{N}}$ to $\mathbb{3}^{\mathbb{N}}$:

Definition 6.1.3. [ᚠ] We define the inclusion map $-^{\uparrow} : \mathbb{2}^{\mathbb{N}} \to \mathbb{3}^{\mathbb{N}}$ that converts representations of $[-1, 1]$ as Cantor sequences $\mathbb{2}^{\mathbb{N}}$ to ternary signed-digit encodings of the same number by mapping every 0 to $\overline{1}$.

Using this map, which is trivially uniformly continuous, we can convert a uniformly continuous function $f : \mathbb{3}^{\mathbb{N}} \to X$, for any closeness space X, into a uniformly continuous function $f^{\uparrow} : \mathbb{2}^{\mathbb{N}} \to X$ with the same modulus of uniform continuity. For our examples, therefore, we usually convert the uniformly continuous predicates and functions on $\mathbb{3}^{\mathbb{N}}$ that we wish to search, optimise and regress into ones on $\mathbb{2}^{\mathbb{N}}$ that we can search, optimise and regress indirectly using uniformly continuous searchers on $\mathbb{2}^{\mathbb{N}}$.

Corollary 6.1.4. [ᚠ] *The Cantor type $\mathbb{2}^{\mathbb{N}}$ is a totally bounded closeness space.*

Proof. [f] Because $\mathbb{2}$ is finite and discrete, the result follows from Corollary 3.2.60.

Corollary 6.1.5. [ᚠ] [ᚠ] *The Cantor closeness space $\mathbb{2}^{\mathbb{N}}$ is uniformly continuously searchable.*

Proof 1. By Corollary 6.1.4 and Theorem 3.3.6.

Proof 2. By Corollary 3.3.12.

$\mathbb{3}^{\mathbb{N}}$ has an approximate linear preorder

A preorder on $\mathbb{3}^{\mathbb{N}}$ is given by the lexicographic order (Lemma 4.1.10), whereas an approximate linear preorder is given by the approximate lexicographic order (Lemma 4.1.17). Recall that these are defined by the following:

$$\begin{aligned}
\alpha \leq_{\mathbb{3}^{\mathbb{N}}} \beta &:= \Pi_{(n\colon \mathbb{N})} \left(\alpha \sim^{n} \beta \to \alpha_n \leq_D \beta_n\right), \\
\alpha \leq^{\varepsilon}_{\mathbb{3}^{\mathbb{N}}} \beta &:= \Pi_{(n\colon \mathbb{N})} \left(n < \varepsilon \to \alpha \sim^{n} \beta \to \alpha_n \leq_D \beta_n\right).
\end{aligned}$$

However, it is not appropriate to use these orders for optimisation and regression on ternary signed-digit encodings, because the lexicographic ordering of the representations of type $\mathbb{3}^{\mathbb{N}}$ does not amount to the numerical ordering of the real numbers of type $\mathbb{I}$. As an example, consider the following different representations $z\alpha, z\beta \colon \mathbb{3}^{\mathbb{N}}$ of $0 \colon \mathbb{I}$:

$$z\alpha := 01\overline{111111}\ldots \text{ and } z\beta := 1\overline{1111111}\ldots.$$

By the above lexicographic order on $\mathbb{3}^{\mathbb{N}}$, it is the case that $z\alpha \leq z\beta$ but $\neg(z\beta \leq z\alpha)$. This issue also propagates to the approximate ordering. Escardó has shown that the lexicographic ordering on signed-digit encodings and the ordering on the encoded numbers can indeed coincide by introducing a normalisation operator $\mathsf{onorm} \colon \mathbb{3}^{\mathbb{N}} \times \mathbb{3}^{\mathbb{N}} \to \mathbb{3}^{\mathbb{N}} \times \mathbb{3}^{\mathbb{N}}$ [Esc98]. The idea is that the normalised pair represents the same numbers (i.e. $\langle\!\langle \alpha \rangle\!\rangle = \langle\!\langle \mathsf{pr}_1(\mathsf{onorm}(\alpha, \beta)) \rangle\!\rangle$ and $\langle\!\langle \beta \rangle\!\rangle = \langle\!\langle \mathsf{pr}_2(\mathsf{onorm}(\alpha, \beta)) \rangle\!\rangle$) but that, on the normalised pair, the lexicographic and numerical orderings coincide.

We take a simpler approach to the problem, and utilise the equivalence between $\mathbb{3}^{\mathbb{N}}$ and the type $\mathbb{T}(-1, 0)_2$ of ternary Boehm encodings used for representing the interval $[-1, 1]$. This equivalence, given in Lemma 5.3.28, yields a 'normalisation' map

$$\mathsf{integer\text{-}approx} \colon \mathbb{3}^{\mathbb{N}} \to (\mathbb{N} \to \mathbb{Z}),$$

where the nth integer k of the sequence $\mathsf{integer\text{-}approx}(\alpha)$, for any $\alpha \colon \mathbb{3}^{\mathbb{N}}$, refers to the nth ternary interval code (Definition 5.3.13) approximation of α — i.e., the interval $\left[\frac{k}{2^n}, \frac{k+2}{2^n}\right]$, in which $\langle\!\langle \alpha \rangle\!\rangle$ lies. Any two signed-digit encodings whose n-prefixes evaluate to the same interval approximations will therefore have the same value of k.

The formal definition of the normalisation map is given in Definition 6.1.6, and

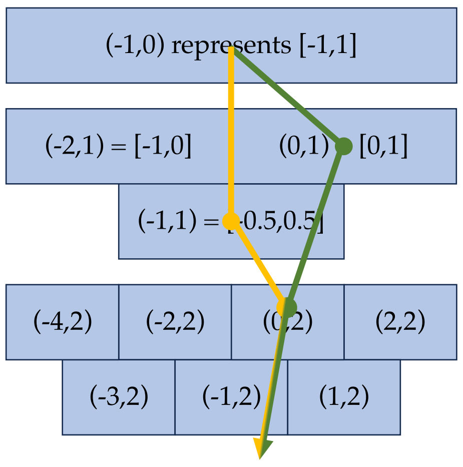


Figure 6.1: Illustration of the interval approximation evaluations of $z\alpha := 01\overline{1}\overline{1}\overline{1}\overline{1}\overline{1}\overline{1}\ldots$ and $z\beta := 1\overline{1}\overline{1}\overline{1}\overline{1}\overline{1}\overline{1}\overline{1}\ldots$ at the first three precision-levels.

its use is informally visualised in Figure 6.1, which shows the relationship between the evaluations of $z\alpha$ and $z\beta$ and their normalised interval approximations. From this illustration, we can read off that

$$\text{integer-approx}(z\alpha) := -1, -1, 0, 0, \ldots \text{ and integer-approx}(z\beta) := -1, 0, 0, 0, \ldots.$$

Definition 6.1.6. [🗲] The *normalisation map* integer-approx: $\mathfrak{3}^{\mathbb{N}} \to (\mathbb{N} \to \mathbb{Z})$, which maps every ternary signed-digit encoding $\alpha : \mathfrak{3}^{\mathbb{N}}$ to a sequence of integers such that, for any $\alpha : \mathfrak{3}^{\mathbb{N}}$, integer-approx$(\alpha)_n$ refers to the n^{th} ternary interval code (Definition 5.3.13) approximation of α is defined as follows:

$$\begin{aligned}
&\mathfrak{3}\text{-to-down} : \mathfrak{3} \to (\mathbb{Z} \to \mathbb{Z}),\\
&\mathfrak{3}\text{-to-down } \overline{1} &&:= \text{downLeft},\\
&\mathfrak{3}\text{-to-down } 0 &&:= \text{downMid},\\
&\mathfrak{3}\text{-to-down } 1 &&:= \text{downRight},
\end{aligned}$$

$$\begin{aligned}
&\text{integer-approx}' : \mathbb{Z} \to \mathfrak{3}^{\mathbb{N}} \to \mathbb{N} \to \mathbb{Z},\\
&\text{integer-approx}'(k, \alpha, 0) &&:= k,\\
&\text{integer-approx}'(k, \alpha, n+1) &&:= \text{integer-approx}'(\mathfrak{3}\text{-to-down}(\text{head } \alpha, k), \text{tail } \alpha, n),
\end{aligned}$$

$$\begin{aligned}
&\text{integer-approx} : \mathfrak{3}^{\mathbb{N}} \to \mathbb{N} \to \mathbb{Z},\\
&\text{integer-approx } \alpha := \text{integer-approx}'(\alpha, -1).
\end{aligned}$$

Using the normalisation map, we define the following preorder and approximate linear preorder on ternary signed-digit encodings $\mathbf{3}^{\mathbb{N}}$, which *are* correct with respect to the numerical ordering on the interval object $\mathbb{I}$. However, the order on $\mathbb{I}$ is not well established, and we have not formalised its notion or verified the correctness of the preorder — we discuss this as further work in Section 7.2.1.

Definition 6.1.7. [✓] We define the *real-order preserving linear preorder* on $\mathbf{3}^{\mathbb{N}}$ by using the normalisation map $\text{integer-approx} : \mathbf{3}^{\mathbb{N}} \to (\mathbb{N} \to \mathbb{Z})$:

$$\begin{aligned} \leq_{\mathbf{3}^{\mathbb{N}}} \quad &: \mathbf{3}^{\mathbb{N}} \to \mathbf{3}^{\mathbb{N}} \to \Omega, \\ \alpha \leq_{\mathbf{3}^{\mathbb{N}}} \beta &:= \exists_{(n :\, \mathbb{N})} \left(\Pi_{(i :\, \mathbb{N})} \left(n \leq i \to \text{integer-approx}(\alpha)_i \leq \text{integer-approx}(\beta)_i\right)\right). \end{aligned}$$

Lemma 6.1.8. [✓] *The real-order preserving preorder is indeed a preorder (Definition 4.1.4).*

Proof. [t] Using propositional truncation (recall this from Section 2.3.6), we prove both reflexivity and transitivity of the real-order preserving preorder:

(i) For any $\alpha : \mathbf{3}^{\mathbb{N}}$, we have that $\text{integer-approx}(\alpha)_i \leq \text{integer-approx}(\beta)_i$ by reflexivity of the order on the integers. Therefore, by setting $n := 0$ we have a proof term $(0, p) : \Sigma_{(n :\, \mathbb{N})} \left(\Pi_{(i :\, \mathbb{N})} \left(n \leq i \to \text{integer-approx}(\alpha)_i \leq \text{integer-approx}(\beta)_i\right)\right)$. The result then follows by truncating this proof term; i.e. $|(0, p)| : \exists_{(n :\, \mathbb{N})} \left(\Pi_{(i :\, \mathbb{N})} \left(n \leq i \to \text{integer-approx}(\alpha)_i \leq \text{integer-approx}(\beta)_i\right)\right)$.

(ii) Given $\alpha, \beta, \zeta : \mathbf{3}^{\mathbb{N}}$ such that $\alpha \leq_{\mathbf{3}^{\mathbb{N}}} \beta$ and $\beta \leq_{\mathbf{3}^{\mathbb{N}}} \zeta$, we wish to show that $\alpha \leq_{\mathbf{3}^{\mathbb{N}}} \zeta$. To do this, we define a function $f : \Sigma_{((n,m) :\, \mathbb{N}\times\mathbb{N})} (\Pi_{(i :\, \mathbb{N})} \left(n \leq i \to \text{integer-approx}(\alpha)_i \leq \text{integer-approx}(\beta)_i\right) \times (m \leq i \to \text{integer-approx}(\beta)_i \leq \text{integer-approx}(\zeta)_i)) \to \exists_{(k :\, \mathbb{N})} (\Pi_{(i :\, \mathbb{N})} (k \leq i \to \text{integer-approx}(\alpha)_i \leq \text{integer-approx}(\zeta)_i))$, and the conclusion is given by the truncation of f. To define this function, we simply set $k := \max(n, m)$, and the proof that $\text{integer-approx}(\alpha)_i \leq \text{integer-approx}(\zeta)_i$ follows by transitivity of the order on the integers.

Definition 6.1.9. [✓] We define the *real-order preserving approximate linear preorder* on $\mathbf{3}^{\mathbb{N}}$ by using the normalisation map $\text{integer-approx} : \mathbf{3}^{\mathbb{N}} \to (\mathbb{N} \to \mathbb{Z})$:

$$\begin{aligned} \leq^{-}_{\mathbf{3}^{\mathbb{N}}} \quad &: \mathbb{N} \to \mathbf{3}^{\mathbb{N}} \to \mathbf{3}^{\mathbb{N}} \to \Omega, \\ \alpha \leq^{n}_{\mathbf{3}^{\mathbb{N}}} \beta &:= \text{integer-approx}(\alpha)_n \leq \text{integer-approx}(\beta)_n. \end{aligned}$$

Lemma 6.1.10. [ ] *The real-order preserving approximate linear preorder is indeed an approximate linear preorder (Definition 4.1.13).*

Proof. The proof that, given any $\varepsilon : \mathbb{N}$, the relation $\leq^{\varepsilon}_{\mathbb{3}^{\mathbb{N}}} : \mathbb{3}^{\mathbb{N}} \to \mathbb{3}^{\mathbb{N}} \to \Omega$ is decidable and a linear preorder is immediate from the fact that the order on the integers is a linear preorder.

To prove that if $C_{\varepsilon}(\alpha, \beta)$ then $\alpha \leq^{\varepsilon}_{\mathbb{3}^{\mathbb{N}}} \beta$ for all $\alpha, \beta : \mathbb{3}^{\mathbb{N}}$, we need to show that $\text{integer-approx}(\alpha)_{\varepsilon} \leq \text{integer-approx}(\beta)_{\varepsilon}$. We first prove the following lemma:

$$\alpha \sim^{\varepsilon} \beta \to \Pi_{(k : \mathbb{Z})} \left(\text{integer-approx}'(k, \alpha, \varepsilon) = \text{integer-approx}'(k, \beta, \varepsilon)\right).$$

We prove this lemma by induction on the given ε. In the base case where $\varepsilon := 0$, then by definition of integer-approx′ (see Definition 6.1.6) we simply have to show that $k = k$. In the inductive case where $\varepsilon := \varepsilon' + 1$ for some $\varepsilon' : \mathbb{N}$, then from $\alpha \sim^{\varepsilon'+1} \beta$ we have that $\text{head } \alpha = \text{head } \beta$ and therefore $\mathbb{3}\text{-to-down}(\text{head } \alpha, k) = \mathbb{3}\text{-to-down}(\text{head } \beta, k)$; the result then follows by the inductive hypothesis.

Now that we have proved the lemma, the result follows from it, Lemma 3.2.59 and the reflexivity of the order on the integers.

Lemma 6.1.11. [ ] *The real-order preserving approximate linear preorder relates (by Definition 4.1.14) to the real-order preserving preorder.*

Proof. [t] The first condition, that if $\alpha \leq_{\mathbb{3}^{\mathbb{N}}} \beta$ then $\exists_{(n : \mathbb{N})} \Pi_{(\varepsilon : \mathbb{N})} (n < \varepsilon \to \alpha \leq^{\varepsilon}_{\mathbb{3}^{\mathbb{N}}} \beta)$, is immediate, because the types are identical. The second condition, that if $x \leq^{n}_{\mathbb{3}^{\mathbb{N}}} y$ holds for all $n : \mathbb{N}$ then $x \leq_{\mathbb{3}^{\mathbb{N}}} y$ follows by truncating the proof term of type $\Sigma_{(n : \mathbb{N})} \Pi_{(\varepsilon : \mathbb{N})} (n < i \to \alpha \leq^{i}_{\mathbb{3}^{\mathbb{N}}} \beta)$, which is constructed by setting $n := 0$ and using the proof $f(i) : x \leq^{i}_{\mathbb{3}^{\mathbb{N}}} y$.

6.1.2 Uniformly continuous exact real arithmetic

In order to perform exact real search on ternary signed-digit encodings, we must prove the uniform continuity of the exact real arithmetic operations. In this subsection, we prove that negation, binary midpoint, infinitary midpoint and multiplication functions (all defined in Section 5.2.3) are uniformly continuous via the closeness space $\mathbb{3}^{\mathbb{N}}$.

Sequence uniform continuity

Recall that discrete-sequence closeness spaces satisfy, for all $n : \mathbb{N}$, $C_n(\alpha, \beta) \Leftrightarrow \alpha \sim^n \beta$ (Corollary 3.2.61); a fact which we use to give the following bespoke definitions of uniform continuity on sequence functions and predicates on discrete types.

Definition 6.1.12. [ ] Given discrete types X and Y, a unary sequence function $f : X^{\mathbb{N}} \to Y^{\mathbb{N}}$ is *uniformly continuous* if for all $\varepsilon : \mathbb{N}$ there is some $\delta : \mathbb{N}$ such that sequences that agree in their δ-prefixes map by f to sequences that agree in their ε-prefixes:

$$\text{seq-f-ucontinuous}^1(f) :=$$
$$\Pi_{(\varepsilon : \mathbb{N})}\Sigma_{(\delta : \mathbb{N})}\Pi_{(x_1, x_2 : X^{\mathbb{N}})}(x_1 \sim^\delta x_2) \to (f(x_1) \sim^\varepsilon f(x_2)).$$

Lemma 6.1.13. [ ] *Given discrete types X and Y, a unary sequence function $f : X^{\mathbb{N}} \to Y^{\mathbb{N}}$ is uniformly continuous by Definition 6.1.12 if and only if it is uniformly continuous by Definition 3.2.27 from the discrete-sequence closeness space yielded by X to the discrete-sequence closeness space yielded by Y.*

Proof. By Corollary 3.2.61.

We next define the binary version of discrete-sequence uniform continuity.

Definition 6.1.14. [ ] Given discrete types X, Y and Z, a binary sequence function $f : X^{\mathbb{N}} \to Y^{\mathbb{N}} \to Z^{\mathbb{N}}$ is *uniformly continuous* if for all $\varepsilon : \mathbb{N}$ there are some $\delta_1, \delta_2 : \mathbb{N}$ such that, given two pairs of sequence arguments, if the first pair agree in their δ_1-prefixes and the second pair agree in their δ_2-prefixes then together they map by f to sequences that agree in their ε-prefixes:

$$\text{seq-f-ucontinuous}^2(f) := \Pi_{(\varepsilon : \mathbb{N})}\Sigma_{((\delta_1, \delta_2) : \mathbb{N} \times \mathbb{N})}\Pi_{(x_1, x_2 : X^{\mathbb{N}})}\Pi_{(y_1, y_2 : Y^{\mathbb{N}})}$$
$$(x_1 \sim^{\delta_1} x_2) \to (y_1 \sim^{\delta_2} y_2) \to (f(x_1, y_1) \sim^\varepsilon f(x_2, y_2)).$$

Note that although the proof of uniform continuity of binary sequence functions by the above definition is logically equivalent to that using the definition of uniform continuity on the binary product (Definition 3.2.42) of two discrete-sequence closeness functions, the former has two moduli of continuity (one for each argument) whereas the latter has only one (i.e. the maximum of the two). This means that the computational content of the proofs differ.

Lemma 6.1.15. [ ] *Given discrete types X, Y and Z, a binary sequence function $f\colon X^{\mathbb{N}} \to Y^{\mathbb{N}} \to Z^{\mathbb{N}}$ is uniformly continuous by Definition 6.1.14 if and only if it is uniformly continuous by Definition 3.2.27 from the binary product of the discrete-sequence closeness spaces yielded by X and Y to the discrete-sequence closeness space yielded by Z.*

Proof. Once again we use the equivalence of $C_n(\alpha, \beta)$ and $\alpha \sim^n \beta$ for all $\alpha, \beta\colon D^{\mathbb{N}}$ (where D is discrete) and $n\colon \mathbb{N}$; i.e. Corollary 3.2.61.

We first show sequence uniform continuity implies closeness uniform continuity. By the former, given $\varepsilon\colon \mathbb{N}, x_1, x_2\colon X^{\mathbb{N}}$ and $y_1, y_2\colon Y^{\mathbb{N}}$, we have that there are $\delta_1, \delta_2\colon \mathbb{N}$ such that if $C_{\delta_1}(x_1, x_2)$ and $C_{\delta_2}(y_1, y_2)$ then $C_\varepsilon(f(x_1, y_1), f(x_2, y_2))$. Therefore we take the modulus of uniform continuity for the closeness space definition to be $\delta := \max(\delta_1, \delta_2)\colon \mathbb{N}$, because (by Lemma 3.2.44 and Corollary 3.2.24) $C_\delta((x_1, y_1), (x_2, y_2))$ implies $C_{\delta_1}(x_1, x_2)$ and $C_{\delta_2}(y_1, y_2)$ and, hence, $C_\varepsilon(f(x_1, y_1), f(x_2, y_2))$ as desired.

The other direction is more straightforward. Given $\varepsilon\colon \mathbb{N}, x_1, x_2\colon X^{\mathbb{N}}$ and $y_1, y_2\colon Y^{\mathbb{N}}$, we have that there is $\delta\colon \mathbb{N}$ such that if $C_\delta((x_1, y_1), (x_2, y_2))$ then $C_\varepsilon(f(x_1, y_1), f(x_2, y_2))$. Therefore we take the moduli of uniform continuity for the sequence definition to be $(\delta, \delta)\colon \mathbb{N} \times \mathbb{N}$, because (by Lemma 3.2.44) $C_\delta((x_1, y_1), (x_2, y_2))$ implies $C_\delta(x_1, x_2)$ and $C_\delta(y_1, y_2)$ and, hence, $C_\varepsilon(f(x_1, y_1), f(x_2, y_2))$ as desired.

Lastly, we define a specialisation of uniform continuity on certain infinitary sequence functions that on discrete types is logically equivalent to uniform continuity on the dependent product (Definition 3.2.42) of a discrete-sequence closeness type.

Definition 6.1.16. [ ] Given discrete types X and Y, an infinitary sequence function $f\colon (X^{\mathbb{N}})^{\mathbb{N}} \to Y^{\mathbb{N}}$ is *uniformly continuous* if for all $\varepsilon\colon \mathbb{N}$ there are $d, \delta\colon \mathbb{N}$ such that infinitary sequences whose d-prefixes agree in their respective δ-prefixes map by f to sequences that agree in their ε-prefixes:

$$\begin{aligned}&\text{seq-f-ucontinuous}^{\mathbb{N}}(f) := \Pi_{(\varepsilon\colon \mathbb{N})}\Sigma_{((d,\delta)\colon \mathbb{N}\times\mathbb{N})}\\ &\qquad \big((d \le \delta) \times (\Pi_{(\alpha,\beta\colon (X^{\mathbb{N}})^{\mathbb{N}})}\Pi_{(n\colon \mathbb{N})}(n < d) \to (\alpha_n \sim^\delta \beta_n) \to (f(\alpha) \sim^\varepsilon f(\beta)))\big).\end{aligned}$$

Lemma 6.1.17. [ ] *Given discrete types X and Y, an infinitary sequence function $f\colon (X^{\mathbb{N}})^{\mathbb{N}} \to Y^{\mathbb{N}}$ is uniformly continuous by Definition 6.1.16 if and only if it is uniformly continuous by Definition 3.2.27 from the countable product of the discrete-sequence closeness space yielded by X to the discrete-sequence closeness space yielded* *by Y.*

Proof. We first require a relationship similar to Corollary 3.2.61 but for sequences $\alpha, \beta : (X^{\mathbb{N}})^{\mathbb{N}}$. Recall from the definition of countable product closeness functions (Definition 3.2.62) and the accompanying visualisation of the diagonalisation argument (Figure 3.1) that for all $\delta : \mathbb{N}$ having $C_\delta(\alpha, \beta)$ means we have $\alpha_0 \sim^{\delta} \beta_0$, $\alpha_1 \sim^{\delta-1} \beta_1$, $\alpha_2 \sim^{\delta-2} \beta_2$, etc.; i.e., for all $n : \mathbb{N}$ such that $n < \delta$ we have $\alpha_n \sim^{\delta-n} \beta_n$. Therefore, we conclude the following relationship:

(i) $C_{2\delta}(\alpha, \beta) \to \Pi_{(n:\,\mathbb{N})} \left(n < \delta \to \alpha_n \sim^{\delta} \beta_n\right)$,

(ii) $\Pi_{(n:\,\mathbb{N})} \left(n < \delta \to \alpha_n \sim^{\delta} \beta_n\right) \to C_\delta(\alpha, \beta)$.

From this relationship, we prove the logical equivalence between the two definitions of uniform continuity.

We first show sequence uniform continuity implies closeness uniform continuity. By the former, given $\varepsilon : \mathbb{N}$, $\alpha, \beta : (X^{\mathbb{N}})^{\mathbb{N}}$ we have that there are $d, \delta' : \mathbb{N}$ such that $d \le \delta'$ and such that if $\alpha_n \sim^{\delta'} \beta_n$ for all $n < d$ then (by Corollary 3.2.61) $C_\varepsilon(f(\alpha), f(\beta))$. To show that closeness uniform continuity is satisfied, we need to give some $\delta : \mathbb{N}$ such that if $C_\delta(\alpha, \beta)$ then $C_\varepsilon(f(\alpha), f(\beta))$. We set $\delta := 2\delta'$, as by (i) this means we have $\alpha_n \sim^{\delta'} \beta_n$ for all $n < \delta'$, and therefore for all $n < d \le \delta'$. By sequence uniform continuity as described, we hence have $C_\varepsilon(f(\alpha), f(\beta))$.

The other direction is again more straightforward. Given $\alpha, \beta : (X^{\mathbb{N}})^{\mathbb{N}}$ we have by closeness uniform continuity that there is some $\delta' : \mathbb{N}$ such that if $C_\delta(\alpha, \beta)$ then $C_\epsilon(f(\alpha), f(\beta))$. To show that sequence uniform continuity is satisfied, we need to give some $d, \delta : \mathbb{N}$ such that $d \le \delta$ and such that if $\alpha_n \sim^{\delta} \beta_n$ for all $n < d$ then $C_\varepsilon(f(\alpha), f(\beta))$. We set both $d, \delta := \delta'$, as by (ii) this means we have $C_\delta(\alpha, \beta)$. By closeness continuity as described, we hence have $C_\varepsilon(f(\alpha), f(\beta))$.

Now we have specialised definitions of uniform continuity for discrete-sequence types which will aid us in proving our operations on $\mathfrak{Z}^{\mathbb{N}}$ are uniformly continuous. Before doing this, we note some expected facts about discrete-sequence uniform continuity.

Lemma 6.1.18. *The identity function on discrete-sequences is uniformly continuous and composition of discrete-sequence functions preserves discrete-sequence uniform continuity.*

Proof (Sketch). Similar to Lemma 3.2.30.

Lemma 6.1.19. [ᚠ] *For any discrete types X and Y, given any function $f : X \to Y$, the function* $\mathsf{map}(f) : X^{\mathbb{N}} \to Y^{\mathbb{N}}$ *is discrete-sequence uniformly continuous.*

Proof. For every $\varepsilon : \mathbb{N}$ the modulus of uniform continuity is ε itself, because $\alpha_\varepsilon = \beta_\varepsilon$ gives $(\mathsf{map}(f, \alpha)_\varepsilon = \mathsf{map}(f, \beta)_\varepsilon) := (f(\alpha_\varepsilon) = f(\beta_\varepsilon))$ using ap.

Lemma 6.1.20. [ᚠ] *For any discrete types X, Y and Z, given any function $f : X \to Y \to Z$, the function* $\mathsf{zipWith}(f) : X^{\mathbb{N}} \to Y^{\mathbb{N}} \to Z^{\mathbb{N}}$ *is discrete-sequence uniformly continuous.*

Proof. Similar to Lemma 6.1.19.

We now explicitly prove the uniform continuity, via the above discrete-sequence definitions (and hence, by the logical equivalences set out above and the discreteness of $\mathbb{3}$, via the necessary closeness space definitions) of the functions neg, mid, bigMid and mul. This task takes up the rest of this subsection.

Negation

First, we prove the uniform continuity of negation.

Corollary 6.1.21. [ᚠ] *Negation on signed-digits encodings is a uniformly continuous function.*

Proof. Recall from Definition 5.2.7 that negation is defined $\mathsf{neg} := \mathsf{map}(\mathsf{flip})$; therefore this is immediately uniformly continuous by Lemma 6.1.19.

Binary midpoint

Next, the uniform continuity of binary midpoint is slightly more complex.

Lemma 6.1.22. [ᚠ] *The function* $\mathsf{div2} : \mathbb{5}^{\mathbb{N}} \to \mathbb{3}^{\mathbb{N}}$ *is uniformly continuous.*

Proof. By its recursive definition (Definition 5.2.13), it can be seen that determining the value of $\mathsf{div2}(\alpha)_\varepsilon$ relies only on α_ε and $\alpha_{\varepsilon+1}$. Therefore, $\varepsilon + 1$ is the modulus of uniform continuity for a given ε.

Corollary 6.1.23. [ᚠ] *Binary midpoint on signed-digit encodings is a uniformly continuous function.*

Proof. Recall from Definition 5.2.15 that binary midpoint is defined $\mathsf{mid}(\alpha,\beta) := \mathsf{div2}(\mathsf{zipWith}(\mathsf{add}\mathbb{3}(\alpha,\beta)))$. The zipWith function is uniformly continuous by Lemma 6.1.20 and the div2 function is uniformly continuous by Lemma 6.1.22. Therefore, the composed function mid is uniformly continuous by Lemma 6.1.18.

Infinitary midpoint

Before we come to multiplication, we must first prove the infinitary midpoint function is uniformly continuous.

Lemma 6.1.24. [✓] *The function* $\mathsf{bigMid}' \colon (\mathbb{3}^{\mathbb{N}})^{\mathbb{N}} \to \mathbb{9}^{\mathbb{N}}$ *is uniformly continuous.*

Proof. By induction on the requested precision $\varepsilon \colon \mathbb{N}$. In the base case, recall from Definition 5.2.27 that $\mathsf{bigMid}'(\zeta)_0$ only uses $(\zeta_0)_0$, $(\zeta_0)_1$ and $(\zeta_1)_0$; therefore $d, \delta := 2$. In the inductive case, recall from Definition 5.2.27 that $\mathsf{bigMid}'(\zeta)_{\varepsilon+1} := \mathsf{bigMid}'(\mathsf{mid}(\mathsf{tail}(\mathsf{tail}\ \zeta_0), \mathsf{tail}\ \zeta_1) :: (\mathsf{tail}(\mathsf{tail}\ \zeta)))$. The argument is the composition of the successor, composition and binary midpoint functions, all of which are uniformly continuous (for binary midpoint, see Corollary 6.1.23); therefore, the function is uniformly continuous by the inductive hypothesis and Lemma 6.1.18.

Lemma 6.1.25. [✓] *The function* $\mathsf{div4} \colon \mathbb{9}^{\mathbb{N}} \to \mathbb{3}^{\mathbb{N}}$ *is uniformly continuous.*

Proof. Similar argument to Lemma 6.1.22.

Corollary 6.1.26. [✓] *Infinitary midpoint on signed-digit encodings is a uniformly continuous function.*

Proof. Recall from Definition 5.2.27 that infinitary midpoint is defined $\mathsf{bigMid} := \mathsf{div4} \circ \mathsf{bigMid}'$. The result follows by Lemmas 6.1.24 and 6.1.25.

Multiplication

Finally, the uniform continuity of multiplication follows from the above.

Corollary 6.1.27. [✓] *Multiplication on signed-digit encodings is a uniformly continuous function.*

Proof. Recall from Definition 5.2.34 that multiplication is defined $\mathsf{mul}(\alpha,\beta) := \mathsf{bigMid}(\mathsf{zipWith}(\mathsf{digitMul},\alpha,\lambda n.\beta))$. This is uniformly continuous by Lemmas 6.1.18 and 6.1.20 and Corollary 6.1.26.

6.1.3 Agda-extracted examples

We have developed, within the TypeTopology library, a large framework of Agda proofs concerning searchable types, generalised optimisation and regression and ternary signed-digit encodings. The true test of this framework is in extracting some proof-of-concept computational algorithms of our generalised framework on $\mathbb{3}^{\mathbb{N}}$. The Agda code is compiled into Haskell as described in Appendix A, which allows it to run faster than if we ran it directly in Agda — though the extracted algorithms are still slow. The reader can try these examples themselves by following the instructions in Appendix A.

Whether we are searching for an answer to a predicate or optimising/regressing a function up to a given precision $\varepsilon : \mathbb{N}$, our algorithms will — using the witness of uniform continuity — effectively compute a finite prefix of a sequence such that any sequence with that prefix will be a correct answer, approximate minimum or satisfactory parameter of the model function.

For each example, we give a table which notes the answer $x : \mathbb{3}^{\mathbb{N}}$ computed for the requested precision-level $\varepsilon : \mathbb{N}$. In the few cases where we search $\mathbb{3}^{\mathbb{N}}$ *directly*[1] each answer is given as a finite prefix and then an ellipses '. . .' to notate the infinitely many 0s our algorithm repeats after the computed prefix. In the usual case where we search $\mathbb{3}^{\mathbb{N}}$ *indirectly*, and hence x is mapped from a sequence of type $\mathbb{2}^{\mathbb{N}}$ using the map defined in Definition 6.1.3), the ellipses '. . .' instead denotes the infinitely-many $\overline{1}$s that the algorithm repeats after the computed prefix. To aid illustration, we also note which real $\langle\!\langle x \rangle\!\rangle : \mathbb{R}$ it is that x represents and we often give further such information, such as the represented value $\langle\!\langle f(x) \rangle\!\rangle$ when we are minimising a particular function f. We note any times above one second taken to compute this answer[2], which grows quickly due to the inefficiency of our underlying arithmetic and the exhaustive nature of our search (further discussions on this folllow in Section 6.2).

Note that, for ease of reading, we abuse notation and often write functions and representations on $\mathbb{3}^{\mathbb{N}}$ as if they are those numbers they represent on the reals $\mathbb{I}$. For example, we write $\frac{1}{4}$ instead of $\overline{1} :: (1 :: (\overline{1} :: \mathsf{repeat}\ 1))$ — but recall that all of these algorithms operate on the representations of the reals.

[1]Recall the discussion on direct and indirect search from Section 6.1.1

[2]For reference, all of our examples are computed using a MacBook Air M1 laptop.

Uniformly continuous search

Search on ternary signed-digit encodings has been performed previously, for example by Escardó in HASKELL [Esc11b]. However, we still provide four examples of uniformly continuous search as proof-of-concept examples of our explicit-continuity assumptions.

Example 6.1.28. [ ] We search for a ternary signed-digit encoding $x : \mathbb{3}^{\mathbb{N}}$ that satisfies

$$p(x) := \frac{-x}{2} \leq^{\varepsilon} \frac{1}{4},$$

for a variety of requested precision values $\varepsilon : \mathbb{N}$ indirectly using the uniformly continuous searcher derived from the totally boundedness of $\mathbb{2}^{\mathbb{N}}$ (i.e. the first proof of Corollary 6.1.5).

The decidability and uniform continuity of the predicate $p : \mathbb{3}^{\mathbb{N}} \to \Omega$ is ensured by that of the approximate linear preorder (Lemma 4.1.15), the uniform continuity of mid (Corollaries 6.1.21 and 6.1.23) and the composition of these (Lemma 3.2.31).

ε	x	$\langle\!\langle x \rangle\!\rangle$	$\frac{-\langle x \rangle}{2}$	Time (s)
5	$1\ldots$	0	0	
10	$1\ldots$	0	0	
15	$1\ldots$	0	0	1.09
20	$1\ldots$	0	0	24.88

Not long after this, the searcher causes a stack overflow. This was not unexpected: recall that the totally bounded searcher in must compute a 2^{δ}-sized δ-net of $\mathbb{2}^{\mathbb{N}}$ in advance of the search.

We next try using the indirect uniformly continuous searcher derived from the decreasing-modulus uniformly continuous searcher of $\mathbb{2}^{\mathbb{N}}$ (i.e. the second proof of Corollary 6.1.5).

ε	x	$\langle\!\langle x \rangle\!\rangle$	$\frac{-\langle x \rangle}{2}$	Time (s)
3	$\overline{1}1\ldots$	−0.5	0.25	
6	$\overline{1}\overline{1}1\ldots$	−0.515625	0.2578125	
9	$\overline{1}\overline{1}11\ldots$	−0.501953125	0.25097656	35.68

This searcher doesn't have the overflow problem, but is in this case less efficient than the totally bounded searcher (likely because the proof of the decreasing-modulus searcher is more computationally expensive). Note also that the two searchers computed different answers; this is because the order in which they evaluate candidate solutions differs — though of course, both are correct up to the requested precision.

The totally bounded searcher in these examples is usually more efficient, but sometimes (due to the difference in search strategy) the decreasing-modulus searcher is better. As in this section we wish to illustrate the correctness of our algorithms, and not their efficiency, from now on we use whichever searcher allows us to produce better results for the given example.

Example 6.1.29. [ ] We search for a ternary signed-digit encoding $x\colon \mathbb{3}^{\mathbb{N}}$ that satisfies

$$p(x) := C_\varepsilon(\mathsf{mul}(x,x), \frac{1}{2}),$$

for a variety of requested precision values $\varepsilon\colon \mathbb{N}$ indirectly using the totally bounded uniformly continuous searcher on $\mathbb{2}^{\mathbb{N}}$.

The decidability and uniform continuity of the predicate $p\colon \mathbb{3}^{\mathbb{N}} \to \Omega$ is ensured by that of the closeness relation (Lemma 3.2.32), the uniform continuity of mul (Corollary 6.1.27) and the composition of these (Lemma 3.2.31).

ε	x	$\langle\!\langle x \rangle\!\rangle$	$\langle\!\langle \mathsf{mul}(x,x) \rangle\!\rangle$	Time (s)
1	$\ldots$	-1	1	
2	$\overline{1}\overline{1}1\ldots$	-0.75	0.5625	
3	$\overline{1}\overline{1}1\ldots$	-0.75	0.5625	
4	$11\overline{1}11\ldots$	0.6875	0.47265625	
5	$\overline{1}\overline{1}1\overline{1}\overline{1}1\ldots$	-0.71875	0.516601563	3.32
6	$11\overline{1}11\overline{1}1\ldots$	0.703125	0.494384766	24.19

The answer correctly converges towards $\pm\sqrt{0.5} = \pm 0.707106781187\ldots$; indeed, it flips between approximations of the two answers for different levels of precision.

At $\varepsilon := 7$, there was a stack overflow. But the decreasing-modulus searcher did not produce an answer in two minutes for $n := 4$. Therefore, this search is much less efficient than that in Example 6.1.28; this is because multiplication requires much higher degrees of input precision than negation and binary midpoint.

Example 6.1.30. [ ] We search for a pair of ternary signed-digit encodings $(x,y)\colon \mathbb{3}^{\mathbb{N}} \times \mathbb{3}^{\mathbb{N}}$ that satisfy

$$p(x,y) := C_\varepsilon(\mathsf{mid}(x,y), 0),$$

for a variety of requested precision values $\varepsilon\colon \mathbb{N}$ indirectly using the totally bounded uniformly continuous searcher on $\mathbb{2}^{\mathbb{N}} \times \mathbb{2}^{\mathbb{N}}$.

The decidability and uniform continuity of the predicate $p \colon \mathbb{3}^{\mathbb{N}} \to \Omega$ is ensured by that of the closeness relation (Lemma 3.2.32), the uniform continuity of mid (Corollary 6.1.23) and the composition of these (Lemma 3.2.31).

ε	x, y	$\langle\!\langle x \rangle\!\rangle, \langle\!\langle y \rangle\!\rangle$	$\langle\!\langle \mathsf{mid}(x,y) \rangle\!\rangle$	Time (s)
5	$11111\ldots$, $\overline{11111}\ldots$	0.9375, -1	-0.03125	9.84
10	$1111111111\ldots$, $\overline{1111111111}\ldots$	0.998046875, -1	-0.000976563	75.5

We have shown we can search for two answers in parallel; although this predicate was particularly well-suited to the search strategy of our exhaustive searcher.

Global optimisation

Using the optimisation algorithm (Theorem 4.1.26), we can optimise any function of ternary signed-digit encodings composed from neg, mid, bigMid and mul — including multivariable and stream functions — to any degree of precision. We give two examples of this.

As global optimisation must evaluate a δ-net of candidates (for a required degree of input precision $\delta \colon \mathbb{N}$ for the requested output precision) in advance of the optimisation process *and* must check each one of these candidates, we find that it very quickly becomes inefficient.

Example 6.1.31. [ᚠ] We compute an ε-global minimum of the function

$$f(x) := \mathsf{neg}(x)$$

for a variety of requested precision values $\varepsilon \colon \mathbb{N}$ indirectly using the totally bounded property of $\mathbb{2}^{\mathbb{N}}$.

The continuity of the function f is by Corollary 6.1.21.

ε	x	$\langle\!\langle x \rangle\!\rangle$	$\langle\!\langle f(x) \rangle\!\rangle$	Time (s)
10	$1111111111\ldots$	0.998046875	-0.999511719	7.68
11	$11111111111\ldots$	0.999023438	-0.999511719	30.23
12	$111111111111\ldots$	0.999511719	-0.999511719	117.18

Example 6.1.32. [ᚠ] We compute an ε-global minimum of the function

$$f(x) := \mathsf{mul}(x, x)$$

for a variety of requested precision values $\varepsilon \colon \mathbb{N}$ indirectly using the totally bounded property of $\mathbb{2}^{\mathbb{N}}$.

The continuity of the function f is by Corollary 6.1.27.

ε	x	$\langle\langle x \rangle\rangle$	$\langle\langle f(x) \rangle\rangle$	Time (s)
1	$\overline{1}\ldots$	-1	1	
2	$1\overline{1}\ldots$	0	0	1.54
3	$1\overline{1}\ldots$	0	0	81.33

Although the same answer is returned each time, the larger ε values means, especially due to the modulus of uniform continuity of exponentiation, an exponentially larger δ-net to exhaust; hence the large jump between the time taken for $\varepsilon := 2$ and $\varepsilon := 3$.

Parametric regression

For regression on ternary signed-digits, we follow the model outlined in Definition 4.2.4. This means that we will be performing regression where the oracle $\mathcal{O}$ is a function and the loss function used is the least-closeness pseudocloseness (Definition 3.2.69) function defined from a given vector $v \colon (\mathbb{3}^{\mathbb{N}})^n$ of n-many predictor observations. Recall that this means the algorithm will only have access to the oracle at the outcomes of the given observations.

Using the regression-as-*optimisation* algorithm (Theorem 4.2.6), we can find an ε-best choice parameter for any uniformly continuous function.

Example 6.1.33. [ᚠ] By fixing the predictor observations $v := \{-1, 0, 1\} \colon (\mathbb{3}^{\mathbb{N}})^3$, we define the least-closeness pseudocloseness function $L_v \colon (X \to Y) \to (X \to Y) \to \mathbb{N}_\infty$ between functions, which compares their values at the points in v. For the oracle function

$$\mathcal{O}(x) := \mathsf{mid}(\frac{1}{3}, x),$$

we compute an ε-best choice parameter $p \colon \mathbb{3}^{\mathbb{N}}$ of the parameterised model function

$$M(p, x) := \mathsf{mid}(\mathsf{neg}(p), x)$$

for a variety of requested precision values $\varepsilon\colon \mathbb{N}$ indirectly by maximising the function $\left(\lambda p.L_v(\mathcal{O}, M(p^{\uparrow}))\right) : 2^{\mathbb{N}} \to \mathbb{N}_\infty$.The shape of the model function matches the oracle function exactly, except for the fact that the parameter is negated – we therefore expect the computed parameter to be close to $-\frac{1}{3}$.

The continuity of the model function is by Corollaries 6.1.21 and 6.1.23 and Lemma 6.1.18.

ε	p	$\langle\langle p \rangle\rangle$	Time (s)
2	$\bar{1}1\bar{1}\dots$	-0.25	
4	$\bar{1}1\bar{1}1\bar{1}\dots$	-0.3125	
6	$\bar{1}1\bar{1}1\bar{1}1\bar{1}\dots$	-0.328125	
8	$\bar{1}1\bar{1}1\bar{1}1\bar{1}1\bar{1}\dots$	-0.33203125	1.26
10	$\bar{1}1\bar{1}1\bar{1}1\bar{1}1\bar{1}1\bar{1}\dots$	-0.333007813	13.79

The optimisation has returned the value that best connects the observations; with this parameter, the regressed function clearly matches the true oracle up to the requested precision.

It is often more practical to use *search* for regression; i.e. to use the algorithms derived from Theorems 4.2.9 and 4.2.10, depending on whether or not there is distortion present in the oracle function.

Example 6.1.34. [ᚠ] For the same predictor observations v, oracle $\mathcal{O}$ and parameterised model function M as in Example 6.1.33, we search for a parameter $p\colon 3^{\mathbb{N}}$ such that

$$\underline{\varepsilon} \preceq L_v(\mathcal{O}, M(p)),$$

for a variety of requested precision values $\varepsilon\colon \mathbb{N}$ indirectly using the totally bounded uniformly continuous searcher on $2^{\mathbb{N}}$.

ε	p	$\langle\langle p \rangle\rangle$	Time (s)
4	$\bar{1}1\bar{1}\dots$	-0.25	
8	$\bar{1}1\bar{1}1\bar{1}1\bar{1}\dots$	-0.328125	2.20
12	$\bar{1}1\bar{1}1\bar{1}1\bar{1}1\bar{1}1\bar{1}\dots$	-0.333007813	9.07
16	$\bar{1}1\bar{1}1\bar{1}1\bar{1}1\bar{1}1\bar{1}1\bar{1}1\bar{1}\dots$	-0.333343506	37.18

Although the parameter is not necessarily ε-best choice, compared to Example 6.1.33 the search routine is quicker and (due to lack of distortion in the oracle) the regressed function still matches the true oracle up to the requested precision.

We continue this example by exploring what happens when we receive outcome observations that are distorted from the true oracle.

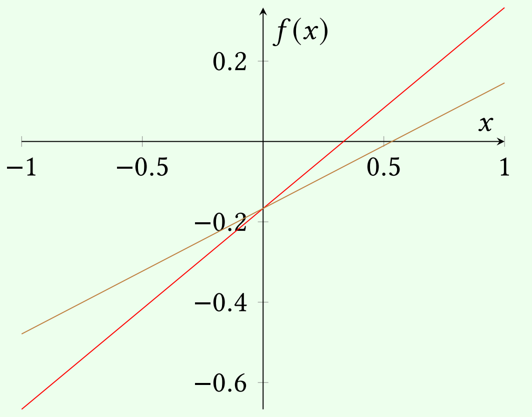
Example 6.1.35. [ᚠ] For the same predictor observations v, oracle $\mathcal{O}$ and parameterised model function M as in Example 6.1.33, we search for a parameter $p\colon \mathfrak{Z}^{\mathbb{N}}$ such that

$$\underline{\varepsilon} \preceq L_v(\Psi(\mathcal{O}), M(p)),$$

directly using the decreasing-modulus uniformly continuous searcher on $\mathfrak{Z}^{\mathbb{N}}$, for a variety of requested precision values $\varepsilon\colon \mathbb{N}$. $\Psi\colon (\mathfrak{Z}^{\mathbb{N}} \to \mathfrak{Z}^{\mathbb{N}}) \to (\mathfrak{Z}^{\mathbb{N}} \to \mathfrak{Z}^{\mathbb{N}})$ is a distortion function that distorts the oracle like so:

$$\Psi(\mathcal{O}) := \lambda x.\mathcal{O}(\mathsf{mid}(x, \frac{1}{4})).$$

The graph below shows the true oracle function $\mathcal{O}$ (in red) plotted against the distorted oracle function $\Psi(\mathcal{O})$ (in brown), the latter of which the searcher can query at points -1, 0 and 1.



ε	p	$\langle\!\langle p \rangle\!\rangle$
1	$0\overline{1}\ldots$	−0.25
2	$0\overline{1}\ldots$	−0.25
3	$1111\ldots$	0.9375

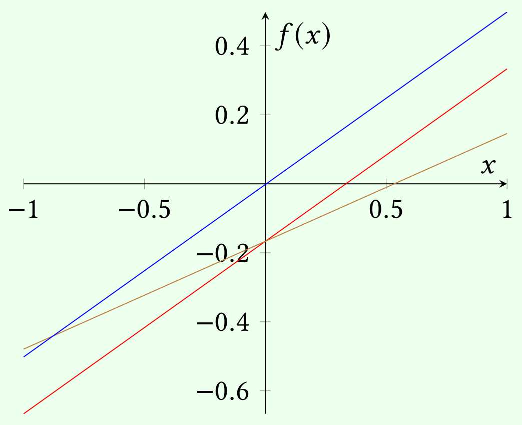
After precision-level $\varepsilon := 2$, the searcher cannot find a parameter p that allows the least-close points (either $M(p, -1)$ and $\Psi(\mathcal{O}(-1))$, $M(p, 0)$ and $\Psi(\mathcal{O}(0))$, or $M(p, 1)$ and $\Psi(\mathcal{O}(1))$) to become ε-close.

However, we could instead simply use optimisation to find the ε-best choice parameter.

Example 6.1.36. [Ɏ] For the same predictor observations v, oracle $\mathcal{O}$, parameterised model function M and distortion function Ψ as in Example 6.1.35, we compute an ε-best choice parameter $p \colon \mathbf{3}^{\mathbb{N}}$ of M for a variety of requested precision values $\varepsilon \colon \mathbb{N}$ directly using the totally bounded property of $\mathbf{3}^{\mathbb{N}}$.

ε	p	$\langle\!\langle p \rangle\!\rangle$	Time (s)
1	$\overline{1}1\dots$	-0.25	
2	$\overline{1}11\dots$	-0.125	
3	$\overline{1}111\dots$	-0.0625	
7	$\overline{1}1111111\dots$	-0.00390625	138.13

The graph below shows the true oracle function $\mathcal{O}$ (in red) plotted against the distorted oracle function $\Psi(\mathcal{O})$ (in brown) and the function $M(p^7)$ (in blue) where p^7 is the p computed when $\varepsilon := 7$, for the interval $[-1, 1]$



Our final example for this section is inspired by linear regression.

Example 6.1.37. [Ɏ] By fixing the predictor observations $v := \{-\frac{1}{2}, \frac{1}{2}\} \colon (\mathbf{3}^{\mathbb{N}})^2$, we define the least-closeness pseudocloseness function $L_v \colon (X \to Y) \to (X \to Y) \to \mathbb{N}_\infty$ between functions, which compares their values at the points in v. We employ the parameterised model function

$$M((p_1, p_2), x) := \mathsf{mid}(p_1, \mathsf{mul}(p_2, x))$$

to search for a parameters $p_1, p_2 \colon \mathbf{3}^{\mathbb{N}}$ such that

$$\underline{\varepsilon} \preceq L_v(\mathcal{O}, M(p_1, p_2)),$$

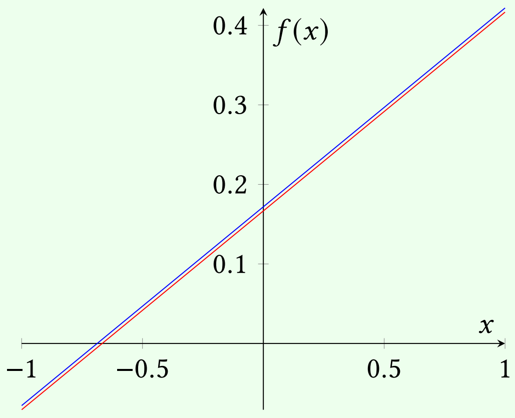
for a variety of precision values $\varepsilon \colon \mathbb{N}$ where $\mathcal{O}$ is the synthetically-constructed oracle function (Definition 4.2.7)

$$\mathcal{O} := M(\frac{1}{3}, -1).$$

The continuity of the model function is by Corollaries 6.1.23 and 6.1.27 and Lemma 6.1.18.

ε	p_1	p_2	$\langle\!\langle p_1 \rangle\!\rangle$	$\langle\!\langle p_2 \rangle\!\rangle$	Time (s)
3	$1\overline{1}\dots$	$11\dots$	0.5	0.5	
4	$1\overline{1}1\overline{1}\dots$	$11\dots$	0.375	0.5	2.30
5	$1\overline{1}1\overline{1}1\overline{1}\dots$	$11\dots$	0.34375	0.5	15.65

The graph below shows the oracle function $\mathcal{O}$ (in red) plotted against the function $M((p_i)^5)$ (in blue) where $(p_i)^5$ are the p_i computed when $\varepsilon := 5$, for the interval $[-1, 1]$



Thus we have concluded that we can indeed perform uniformly continuous search and generalised global optimisation and parametric regression on ternary signed-digits. The correctness of the algorithms are immediate, because they are extracted from our formal Agda framework, but we enjoyed illustrating this fact. Unfortunately, but not unexpectedly, the efficiency of the algorithms leaves a lot to be desired.

6.2 Exact Real Search using ternary Boehm encodings

The signed-digits have provided a proof-of-concept for applying our generalised perspective of optimisation and regression to types for representing real numbers. Furthermore, we have verified the signed-digits, so that we are genuinely optimising/regressing representations of functions on the compact interval we are searching.

There are, however, clear problems with using signed-digits for potential practical applications of exact real search, which can be summed up in two points. Firstly, the arithmetic defined on the signed-digits is not user-friendly in the same way that, say, arithmetic on the floating-point numbers are — we cannot even perform addition using signed-digit numbers without utilising multiple representations of different intervals. Secondly, arithmetic and search on the signed-digits is inefficient; in particular, determining the n-approximation of a represented number requires the evaluation of the whole n-prefix of the sequence.

The ternary Boehm encodings address both of these problems. For the former, the Boehm encodings represent reals across the real line, meaning we can immediately perform arithmetic operations such as addition that are not suitable for compact intervals. For the latter, although the arithmetic still remains inefficient, it is much more eficient than that on the ternary signed-digits, and leads us towards much more practical algorithms. Further, recall that the structure of the ternary Boehms means that the n^{th} interval approximation of a represented number can be determined by evaluating *exactly* the n^{th} integer approximation of the sequence.

We turn our attention in this final section, therefore, towards the potential of practical search, optimisation and regression using the ternary Boehm encodings. From this point on we depart from absolute guarantees of correctness in favour of investigating whether our framework can lead us towards a practical implementation of exact real search. Although we find a positive answer, there is much work to be done to actually deliver on this desire — and, further still, to tie it in with guarantees of correctness, which we discuss as further work in Section 7.2.3.

6.2.1 Suitability for search, optimisation and regression

Recall that, in Section 5.3, we re-rationalised Boehm's encodings in our formal framework as the type $\mathbb{T}$ and prepared them for search by defining the subtypes $\mathbb{T}(k,i)_i$ (for $i \in \{1,2,3\}$ for representing real numbers in compact intervals $\left[\frac{k}{2^i}, \frac{k+2}{2^i}\right]$ encoded as pairs $(k,i) : \mathbb{Z}^2$.

$\mathbb{T}$ yields continuously searchable closeness spaces

We can easily show that any type $\mathbb{T}(k,i)_2$ is suitable for search, optimisation and regression because we have already proved its equivalence with $\mathfrak{Z}^{\mathbb{N}}$ (Lemma 5.3.28), a type we recently proved is suitable for these processes (in Section 6.1.1). Therefore, we can search $\mathbb{T}$ in the same way as we directly search $\mathfrak{Z}^{\mathbb{N}}$.

Corollary 6.2.1. $\mathbb{T}(k,i)_2$ *is a totally bounded, uniformly continuously searchable closeness space.*

Proof. [f] By Corollary 6.1.1, via the equivalence with $\mathbb{3}^{\mathbb{N}}$ (Lemma 5.3.28).

We can instead search $\mathbb{T}(k,i)_3$ in the same way as we *indirectly* search $\mathbb{3}^{\mathbb{N}}$ using $\mathbb{2}^{\mathbb{N}}$. This is because — recalling the structural operations from Section 5.3 — we only need to consider those interval approximations that are recursively downLeft or downRight from $(k,i)\colon \mathbb{Z}^2$ in order to to determine an answer for any of our algorithms (i.e. we do not need to use downMid). In our Java library, we effectively search elements of $\mathbb{T}(k,i)_3$ as described in Section 6.2.2.

$\mathbb{T}$ yields approximate linear preorders

Recall that in order to determine the n^{th} interval approximation of some $x\colon \mathbb{T}$, rather than evaluating the whole n-prefix, it is enough to evaluate $x_n\colon \mathbb{Z}$. This means that we can immediately compare these interval approximations in a more convenient way than the order and closeness functions are used for searching $\mathbb{3}^{\mathbb{N}}$. For example, instead of a predicate asking whether $C_\varepsilon(x,y)$, requiring us to evaluate x and y up to ε in order to find out (by Remark 3.2.20) whether or not $d_{\mathbb{R}}([\![x]\!],[\![y]\!]) < 2^{-\varepsilon}$, we can instead ask whether $|x_\varepsilon - y_\varepsilon| \leq 1$. This further means our representations do not have to match at every point up to ε, only at ε itself.

Another consequence of this direct evaluation is for comparing the order of elements of $\mathbb{T}$. Recall that, for $\mathbb{3}^{\mathbb{N}}$, we had to introduce a new approximate linear preorder (Definition 6.1.9) which evaluates prefixes of $\mathbb{3}^{\mathbb{N}}$ and converts them to ternary interval codes so that they can be compared. Elements of $\mathbb{T}$ are trivial to convert to ternary interval codes at any point, and therefore we can directly compare the interval approximations without any need for exhaustive evaluation.

6.2.2 Java-implemented examples

We have written, in Java, an implementation of the ternary Boehm encodings that allows us to perform search, optimisation and regression. The implementation is based on the re-rationalisation of ternary Boehm encodings in our Agda library (described in Section 5.3) as well as on our informal discussions in the previous section. We define arithmetic operations on $\mathbb{T}$ by completing approximations of them defined on dyadic interval codes $\mathbb{Z}^2$ (Definition 5.3.11). A base operation's modulus of uniform continuity is hard-coded into the function object, and composing functions builds a new modulus of uniform continuity from its constitutent functions'.

The Java implementation is outlined in Appendix B; here we just give a brief idea of the algorithms, which are effectively the same as the indirect algorithms on $\mathbb{2}^{\mathbb{N}}$ that we used to search $\mathfrak{Z}^{\mathbb{N}}$. For the compact interval represented by $(k, i) : \mathbb{Z}^2$, whether we are searching for an answer to a predicate or optimising/regressing a function up to a given precision $\varepsilon : \mathbb{Z}$, our implementation will — using the witness of uniform continuity $\delta : \mathbb{Z}$ — search the finitely-many search candidates that are recursively downLeft or downRight of (k, i) on precision-level δ. Each candidate (c, δ) is then cast to a ternary Boehm encoding $x : \mathbb{T}$ such that $x_\delta = c$, which can be tested against the predicate (or passed to the function being optimised). In this way, the algorithms compute an interval approximation of a ternary Boehm encoding such that any element of $\mathbb{T}(k, i)_3$ that features that interval approximation will be a correct answer, approximate minimum or satisfactory parameter of the model function.

For each example, we give a table which notes the computed interval approximation $(x_\delta, \delta) : \mathbb{Z}^2$ of the answer $x : \mathbb{T}(k, i)_3$ for the requested output precision-level ε, required input-precision level δ and searched compact interval (k, i). To aid illustration, we also note the dyadic $\frac{k+1}{2^i}$ at the center of the interval that (k, i) represents, and we often give further such information, such as the represented value $\frac{f(\varepsilon)+1}{2^\varepsilon}$ when we are minimising a particular function f.

We note any times above one second taken to compute this answer, and note that we are broadly more efficient than on the ternary signed-digits. However, as the nature of the search is still exhaustive, the improvements are not seismic. We discuss the ability to perform branch-and-bound style optimisation (and search) techniques, to further increase efficiency, in Section 6.2.3.

Uniformly continuous search

Example 6.2.2. This example is based on Example 6.1.29. We search for a ternary Boehm encoding $x : \mathbb{T}$ in $[-1, 1]$ that satisfies

$$p(x) := \text{abs}\left(\left(x^2\right)_{\varepsilon+1} - \left(\frac{1}{2}\right)_{\varepsilon+1}\right) \leq 1,$$

for a variety of requested precision values $\varepsilon : \mathbb{Z}$.

ε	(x_δ, δ)	$\frac{x_\delta+1}{2^\delta}$	$\frac{(x^2)_\varepsilon+1}{2^\varepsilon}$	Time (s)
5	$(-96, 7)$	−0.7421875	0.550842285	
10	$(-2902, 12)$	−0.708251953	0.501620829	
15	$(-92688, 17)$	−0.707145691	0.500055028	
20	$(-2965826, 22)$	−0.707107782	0.500001416	1.03
25	$(-94906272, 27)$	−0.707106821	0.500000057	32.4

The answer correctly converges towards the irrational number $-\sqrt{0.5} = -0.707106781187\ldots$. It could have alternatively converged towards $\sqrt{0.5}$, but our searcher evaluates candidates in ascending integer approximation order.

Due to the efficiency gains of ternary Boehm encodings, along with the better (but not formally verified) modulus of uniform continuity on multiplication, we are able to compute the answer to a much higher precision-level than we could in Example 6.1.29.

Example 6.2.3. This example is based on Example 6.1.30. We search for ternary Boehm encodings $x, y \colon \mathbb{T}$ in $[-1, 1]$ that satisfy

$$p(x) := \mathsf{abs}\left(\mathsf{mid}(x, y)_{\varepsilon+1} - 0_{\varepsilon+1}\right) \leq 1,$$

for a variety of requested precision values $\varepsilon \colon \mathbb{Z}$.

ε	(x_{δ_1}, δ_1)	(y_{δ_2}, δ_2)	$\frac{x_{\delta_1}+1}{2^{\delta_1}}$	$\frac{y_{\delta_1}+2}{2^{\delta_2}}$	Time (s)
5	$(-256, 8)$	$(242, 8)$	-0.99609375	0.94921875	
10	$(-8192, 13)$	$(8178, 13)$	-0.99987793	0.998413086	
15	$(-262144, 18)$	$(262130, 18)$	-0.999996185	0.999950409	
20	$(-8388608, 23)$	$(8388594, 23)$	-0.999999881	0.99999845	
25	$(-268435456, 28)$	$(268435442, 28)$	-0.999999996	0.999999952	356.72

This answer computes quickly to a reasonably high degree of precision; although, as we noted in Example 6.1.30, the predicate is particularly well-suited to the search strategy of our exhaustive searcher.

Example 6.2.4. This example is of a predicate not well-suited to our exhaustive searcher, and in a different interval than $[-1, 1]$.

We tried search for a ternary Boehm encoding $x \colon \mathbb{T}$ in $[16, 24]$ that satisfies

$$p(x) := x^3 + 3x \geq^{\varepsilon} 9000,$$

for a variety of requested precision values $\varepsilon \colon \mathbb{Z}$. Unfortunately, even for $\varepsilon := 1$, the search process hanged for over two minutes. Clearly the number of search candidates at the level of input precision required is too great for an efficient result to be returned. We will revisit this example later, in Example 6.2.11.

Global optimisation

We define the optimisation algorithm on ternary Boehm encodings based on that arising from Theorem 4.1.26; it computes the δ-net of interval approximations of $\mathbb{T}(k,i)_3$, where $\delta : \mathbb{Z}$ is the modulus of uniform continuity of the function being optimised and (k,i) is the interval being searched for an ε-global minimum.

Example 6.2.5. This example is based on Example 6.1.32. We compute an ε-global minimum of the function

$$f(x) := x * -1$$

in $[-1,1]$ for a variety of requested precision values $\varepsilon : \mathbb{Z}$.

ε	(x_δ, δ)	$\frac{x_\delta+1}{2^\delta}$	$\frac{(-x)_\varepsilon+1}{2^\varepsilon}$	Time (s)
1	$(14, 4)$	0.9375	−0.9375	
2	$(62, 6)$	0.984375	−0.984375	
3	$(254, 8)$	0.99609375	−0.99609375	
4	$(1022, 10)$	0.999023438	−0.999023438	
5	$(4094, 12)$	0.999755859	−0.999755859	
6	$(16382, 14)$	0.999938965	−0.999938965	
7	$(65534, 16)$	0.999984741	−0.999984741	
8	$(262142, 18)$	0.999996185	−0.999996185	4.45
9	$(1048574, 20)$	0.999999046	−0.999999046	97.94

This problem was designed specifically so that we had to nearly exhaust the net; hence, it takes a long time to compute. Compared to Example 6.1.32, we are only able to compute slightly more precise approximations. This is because, although multiplication's modulus of uniform continuity and the structure of the ternary Boehm encodings admit more efficiency than those on ternary signed-digit encodings, the fact that we have to exhaust the δ-net cannot be avoided. We will revisit this example later, in Example 6.2.12.

Example 6.2.6. The efficiency issues seen in Example 6.2.5 become worse with a more complicated function. We tried to compute an ε-global minimum of the function

$$f(x) := x^6 + x^5 - x^4 + x^2$$

in $[-2,2]$ (illustrated in Figure 4.1) for a variety of requested precision values $\varepsilon : \mathbb{Z}$. Unfortunately, even for $\varepsilon := 1$, the search process hanged for over two minutes.

Clearly the number of candidates in the δ-net is too great for an result to be returned in reasonable time. We will revisit this example later, in Example 6.2.13.

Parametric regression

For regression on ternary Boehm encodings, we follow the rough idea of the model outlined in Definition 4.2.4 and used in Section 6.1.3. However, as we now have access to addition, we can tweak our loss function to be more practical. Rather than returning the least-closeness value $\mathsf{min}(c(M_p(x_0), \mathcal{O}(x_0)), ..., c(M_p(x_{n-1}), \mathcal{O}(x_{n-1})))$, for model function M, oracle $\mathcal{O}$, parameter choice p and observations $x_0, ..., x_{n-1}$, we instead simply sum the distances[3]:

$$\sum_{i:=0}^{n} \mathsf{abs}(M_p(x_i) - \mathcal{O}(x_i)).$$

By implementing the regression-as-*optimisation* algorithm (Theorem 4.2.6), we can find an ε-best choice parameter for any uniformly continuous function.

Example 6.2.7. This example is based on Example 6.1.33. By fixing the predictor observations $v := \{-1, 0, 1\} : (\mathbb{T})^3$, we define the absolute loss function $L_v : (\mathbb{T} \to \mathbb{T}) \to (\mathbb{T} \to \mathbb{T}) \to \mathbb{T}$ between functions, which sums the difference of their values at the points in v. For the oracle function

$$\mathcal{O}(x) := \mathsf{mid}(\frac{1}{3}, x),$$

we compute an ε-best choice parameter $p : \mathbb{T}$ in $[-1, 1]$ of the parameterised model function

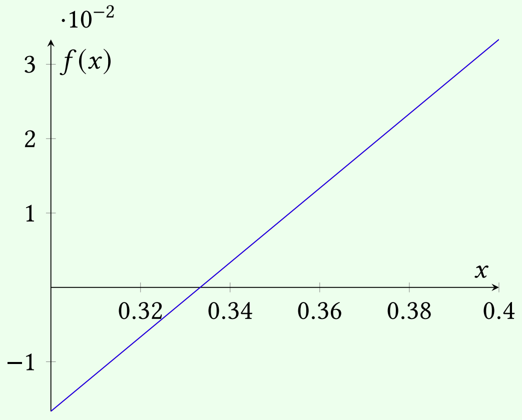
$$M(p, x) := \mathsf{mid}(-p, x)$$

for a variety of requested precision values $\varepsilon : \mathbb{Z}$ by minimising the function $(\lambda p.L_v(\mathcal{O}, M(p))) : \mathbb{T}(-1, 0)_3 \to \mathbb{T}$ in the interval $[-1, 1]$.

[3]This is similar to using the least-squares loss function, a common loss function for parametric regression.

ε	(p_δ, δ)	$\frac{p_\delta+1}{2^\delta}$	Time (s)
3	$(-84, 8)$	-0.25	
6	$(-682, 11)$	-0.328125	
9	$(-5460, 14)$	-0.33203125	
12	$(-43690, 17)$	-0.333251953125	4.72
15	$(-349524, 20)$	-0.33331298828125	103.59

The graph below shows the oracle function $\mathcal{O}$ (in red) plotted against the function $M(p^{15})$ (in blue) where p^{15} is the p computed when $\varepsilon := 15$, for the interval $[0.3, 0.4]$. It is hard to tell the two lines apart due to the closeness of the true parameter and the regressed parameter.



We can see that the output precisions are very close, and that the efficiency is much improved when using ternary Boehm, encodings. We will revisit this example later, in Example 6.2.14.

As we have discussed, it is often more practical to use our regression algorithms that are derived from our searchers.

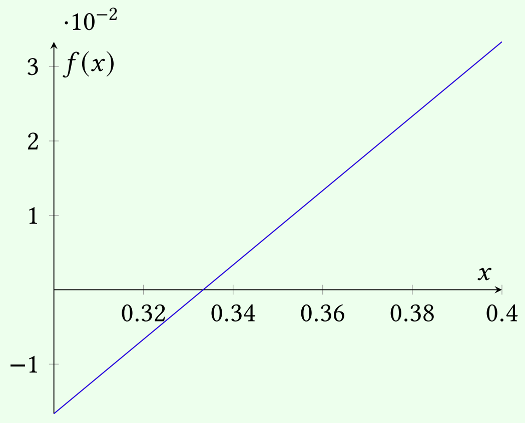
Example 6.2.8. This example is based on Example 6.1.34. For the same predictor observations v, oracle $\mathcal{O}$ and parameterised model function M as in Example 6.1.33, we search for a parameter $p\colon \mathbb{T}$ in $[-1, 1]$ such that

$$L_v(\mathcal{O}, M(p))_\varepsilon \leq 1,$$

for a variety of requested precision values $\varepsilon\colon \mathbb{N}$.

ε	(p_δ, δ)	$\frac{p_\delta+1}{2^\delta}$	Time (s)
3	$(-218, 9)$	-0.42578125	
6	$(-1412, 12)$	-0.3447265625	
9	$(-10970, 15)$	-0.33477783203125	1.76
12	$(-87428, 18)$	-0.3335113525390625	20.2
15	$(-699098, 21)$	-0.3333559036254883	161.84

The graph below shows the oracle function $\mathcal{O}$ (in red) plotted against the function $M(p^8)$ (in blue) where p^8 is the p computed when $\varepsilon := 8$, for the interval $[0.3, 0.4]$

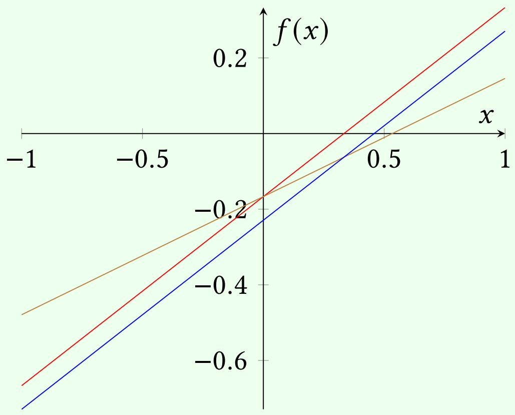


We found that with the ternary signed-digit encodings, the search version of this example (i.e. Example 6.1.34) was more efficient than the optimisation version (Example 6.1.33). Interestingly, we find that this is reversed when using the ternary Boehm encodings (i.e. this example is less efficient than Example 6.2.7).

Example 6.2.9. This example is based on Example 6.1.36. For the same predictor observations v, oracle $\mathcal{O}$ and parameterised model function M as in Example 6.2.8, we compute an ε-best choice parameter $p : \mathbb{T}$ of M for a variety of requested precision values $\varepsilon : \mathbb{Z}$ by minimising the function $(\lambda p.L_v(\Psi(\mathcal{O}), M(p))) : \mathbb{T}(-1,0)_3 \to \mathbb{T}$ in the interval $[-1, 1]$.

ε	(p_δ, δ)	$\frac{p_\delta+1}{2^\delta}\rangle\!\rangle$	Time (s)
3	$(-114, 8)$	-0.44140625	
6	$(-936, 11)$	-0.456542969	
9	$(-7506, 14)$	-0.458068848	
12	$(-60072, 17)$	-0.458305359	5.42
15	$(-480594, 21)$	-0.458329201	110.76

The graph below shows the true oracle function $\mathcal{O}$ (in red) plotted against the distorted oracle function $\Psi(\mathcal{O})$ (in brown) and the function $M(p^{15})$ (in blue) where p^{15} is the p computed when $\varepsilon := 15$, for the interval $[-1, 1]$



Our final example for this subsection is an instantiation of linear regression.

Example 6.2.10. This example is based on Example 6.1.37 (but with binary midpoint replaced with addition). By fixing the predictor observations $v := \{-\frac{1}{2}, \frac{1}{2}\} \colon (\mathbb{T})^2$, we define the absolute loss function $L_v \colon (\mathbb{T} \to \mathbb{T}) \to (\mathbb{T} \to \mathbb{T}) \to \mathbb{T}$ between functions, which sums the difference of their values at the points in v. We employ the parameterised model function

$$M((p_1, p_2), x) := p_1 + p_2 * x$$

to search for parameters $p_1, p_2 \colon \mathbb{T}$ in $[-1, 1]$ such that

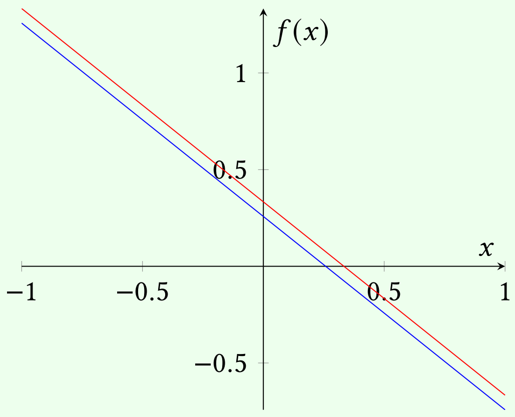
$$L_v(\mathcal{O}, M(p_1, p_2))_\varepsilon \leq 1,$$

for a variety of precision values $\varepsilon \colon \mathbb{Z}$ where $\mathcal{O}$ is the synthetically-constructed oracle function (Definition 4.2.7)

$$\mathcal{O} := M(\frac{1}{3}, -1).$$

ε	$((p_1)_{\delta_1}, \delta_1)$	$((p_2)_{\delta_2}, \delta_2)$	$\frac{(p_1)_{\delta_1}+1}{\delta_1}$	$\frac{(p_2)_{\delta_2}+1}{\delta_2}$	Time (s)
1	$(0, 5)$	$(-2048, 11)$	0.03125	−0.999511719	1.3
2	$(10, 6)$	$(-8192, 13)$	0.171875	−0.99987793	9.50
3	$(32, 7)$	$(-32768, 15)$	0.2578125	−0.999969482	86.52

The graph below shows the oracle function $\mathcal{O}$ (in red) plotted against the function $M(p^3)$ (in blue) where p^3 is the p computed when $\varepsilon := 3$, for the interval $[-1, 1]$



The computation when fitting a linear model to a correct oracle, even in a simple case, is still remarkably inefficient. We will revisit this example later, in Example 6.2.15.

6.2.3 Java-implemented branch-and-bound examples

The ternary Boehms are clearly a much more efficient data-type than the ternary signed-digits for performing exact real search, optimisation and regression on functions for exact real arithmetic. However, the exhaustive nature of each algorithm means that the efficiency of the search relies heavily on how coincidentally well-suited the predicate or function is to the order in which the algorithms evaluates the candidate interval approximations. In this final subsection, we discuss how *branch-and-bound* techniques can be defined on the ternary Boehm encodings, and give examples of their use from our Java implementation.

Branch-and-bound algorithms are a popular and well-studied technique in optimisation theory that can improve the efficiency of exhaustive search by discarding any candidates that outright cannot contain a solution [Cla99; BM07]. A (unary) branch-and-bound algorithm works as follows:

1. *Initialise* the search area as some candidate interval,
2. *Select* some candidate from the search area using heuristic criteria,
3. *Branch* the candidate into multiple sub-intervals,
4. *Bound* the sub-intervals by computing each of their lower and upper bounds of f,
5. *Discard* any candidates in the search area that cannot contain a global minimiser (i.e. those whose lower bound is strictly higher than another candidate's upper bound),
6. Repeat from step (2) until the width of the range of the search space is less than the desired precision; then return any remaining candidate interval.

In the next iteration, the potential solution will either be the same width or thinner than the current. After each iteration, the remaining search area contains a solution to the global minimisation problem; furthermore, if the width of the remaining search area's output is less than the desired precision ε, then it is a solution to the ε-global minimisation problem.

Branch-and-bound algorithms converge given (i) the function f is continuous, (ii) the *branching* procedure ensures the width of the widest interval tends to 0, and (iii) the *bounding* procedure ensures that the distance between the lower and upper-bound estimates for each interval also tends to 0 [Kea92].

For ternary Boehm encodings, the branching procedure is the dissection of a ternary interval code $(k, i) : \mathbb{Z}^2$ into the two intervals $(\mathsf{downLeft}\ k, i{+}1), (\mathsf{downRight}\ k, i{+}1) : \mathbb{Z}^2$ directly below it. The bounding procedure, meanwhile, is the use of the function's modulus of uniform continuity to bound its behaviour on interval codes.

We postulate that the ternary Boehm encodings can be used to satisfy all three of these conditions:

(i) The functions we define are indeed uniformly continuous on the intervals on which we search them,

(i) The branching procedure halves the width of the candidate interval approximations and — as there are finitely-many candidate intervals that are only branched up to the precision-level given by the modulus of uniform continuity — the widest interval will be divided in finite time,

(i) The bounding procedure of the straightforward functions we have defined (in Sections 5.3.3 and 7.2.2) will decrease the width of the output intervals as the width of the input intervals decreases.

We aim in future work to define this class of algorithms formally in our library and verify their convergence (see Section 7.2.3); for now, we give informal JAVA implementations of their use.

Uniformly continuous search

Example 6.2.11. We revisit Example 6.2.4, an example where we previously could not even compute an answer for output precision-level $\varepsilon := 1$. We search for a ternary Boehm encoding $x : \mathbb{T}$ in $[16, 24]$ that satisfies

$$p(x) := x^3 + 3x \geq^{\varepsilon} 9000,$$

for a variety of requested precision values $\varepsilon : \mathbb{Z}$ using the branching searcher.

ε	(x_δ, δ)	$\frac{x_\delta+1}{2^\delta}$	$\frac{(x^3+3x)_\varepsilon+1}{2^\varepsilon}$
50	$(42, 1)$	21	9324
100	$(42, 1)$	21	9324
150	$(42, 1)$	21	9324
200	$(42, 1)$	21	9324

Our branching searcher appears almost to have cheated. Previously, the number of search candidates at the level of input precision required was too great for an efficient result to be returned. But, using the branching searcher, we find that the input precision required is in fact very low: the predicate is satisfied easily by any interval approximation that gives an output in the upper half of the search area.

Global optimisation

Example 6.2.12. Revisiting Example 6.1.32, we compute an ε-global minimum of the function

$$f(x) := x * -1$$

in $[-1, 1]$ for a variety of requested precision values $\varepsilon : \mathbb{Z}$ using the branch-and-bound technique.

ε	(x_δ, δ)
50	$(1125899906842622, 50)$
100	$(1267650600228229401496703205374, 100)$
150	$(1427247692705959881058285969449495136382746622, 150)$
200	$(1606938044258990275541962092341162602522202993782792835301374, 200)$

ε	$\frac{x_\delta+1}{2^\delta}$	$\frac{(-x)_\varepsilon+1}{2^\varepsilon}$
50	0.9999999999999982	-0.9375
100	≈ 1.0	≈ -1.0
150	≈ 1.0	≈ -1.0
200	≈ 1.0	≈ -1.0

Originally, this problem was designed specifically so that we had to nearly exhaust the net. Using branch-and-bound, this no longer occurs; the simple linear shape of the function allows the algorithm to quickly (in less than 10ms in all above cases) 'zoom in' on the solution.

Example 6.2.13. We revisit Example 6.2.6, an example where we previously could not even compute an answer for output precision-level $\varepsilon := 1$. We compute an ε-global minimum of the function

$$f(x) := x^6 + x^5 - x^4 + x^2$$

in $[-2, 2]$ (illustrated in Figure 4.1) for a variety of requested precision values $\varepsilon\colon \mathbb{Z}$ using the branch-and-bound technique.

ε	(x_δ, δ)	$\frac{x_\delta+1}{2^\delta}$	$\frac{(f(x)_\varepsilon+1}{2^\varepsilon}$
5	$(-236, 8)$	-0.90625	-0.0625
10	$(-7384, 13)$	-0.9013671875	-0.044921875
15	$(-236058, 18)$	-0.900482177734375	-0.043670654296875
20	$(-7553656, 23)$	-0.9004659652709961	-0.04366016387939453
25	$(-241716810, 28)$	-0.9004652798175812	-0.043659746646881104

The issue with the exhaustive algorithm was that the δ-net quickly became so granular, due to the complex shape of the function, that there were far too many candidates to search efficiently. The branch-and-bound technique helps to tackle this problem: it quickly discards intervals that cannot contain the minimum, refining the search space so that we find an answer much more efficiently.

Parametric regression

Example 6.2.14. Revisiting Example 6.2.7, we fix the predictor observations $v := \{-1, 0, 1\}\colon (\mathbb{T})^3$ and define the absolute loss function $L_v\colon (\mathbb{T} \to \mathbb{T}) \to (\mathbb{T} \to \mathbb{T}) \to \mathbb{T}$ between functions, which sums the difference of their values at the points in v. For the oracle function

$$\mathcal{O}(x) := \mathsf{mid}(\frac{1}{3}, x),$$

we compute an ε-best choice parameter $p\colon \mathbb{T}$ in $[-1, 1]$ of the parameterised model function

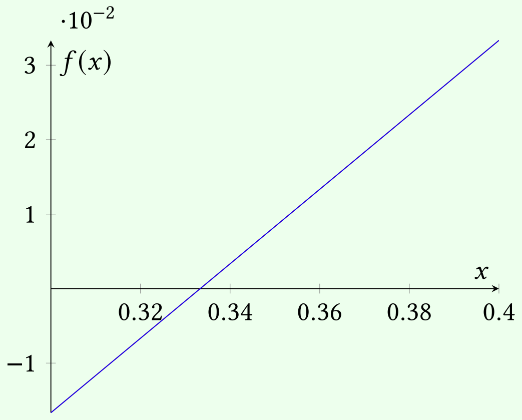
$$M(p, x) := \mathsf{mid}(-p, x)$$

for a variety of requested precision values $\varepsilon\colon \mathbb{Z}$ by minimising the function $(\lambda p.L_v(\mathcal{O}, M(p)))\, :\, \mathbb{T}(-1, 0)_3 \to \mathbb{T}$ in the interval $[-1, 1]$ using the branch-and-bound technique.

ε	(p_δ, δ)
50	$(-1501199875790164, 52)$
100	$(-1690200800304305868662270940500, 102)$
150	$(-1902996923607946508077714625932660181843662164, 152)$
200	$(-2142584059011987034055949456454883470029603991710390447068500, 202)$

ε	$\frac{p_\delta+1}{2^\delta}$
50	-0.333333333333333304
100	$\approx -0.333333333333333333\ldots$
150	$\approx -0.333333333333333333\ldots$
200	$\approx -0.333333333333333333\ldots$

The graph below shows the oracle function $\mathcal{O}$ (in red) plotted against the function $M(p^{200})$ (in blue) where p^{200} is the p computed when $\varepsilon := 200$, for the interval $[0.3, 0.4]$



The efficiency is hugely improved by using the branch-and-bound technique.

Example 6.2.15. Revisiting Example 6.2.10, we fix the predictor observations $v := \{-\frac{1}{2}, \frac{1}{2}\} \colon (\mathbb{T})^2$ and define the absolute loss function $L_v \colon (\mathbb{T} \to \mathbb{T}) \to (\mathbb{T} \to \mathbb{T}) \to \mathbb{T}$ between functions, which sums the difference of their values at the points in v. We employ the parameterised model function

$$M((p_1, p_2), x) := p_1 + p_2 * x$$

to search for parameters $p_1, p_2 \colon \mathbb{T}$ in $[-1, 1]$ such that

$$L_v(\mathcal{O}, M(p_1, p_2))_\varepsilon \leq 1,$$

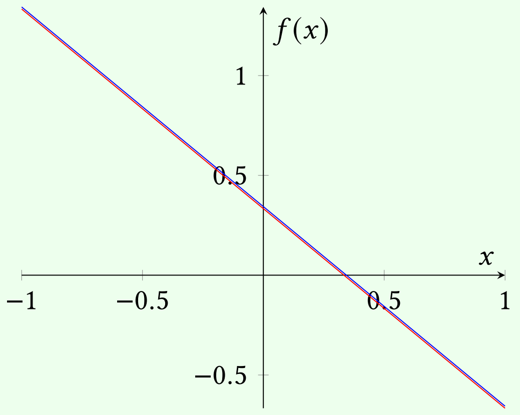
for a variety of precision values $\varepsilon \colon \mathbb{Z}$, where $\mathcal{O}$ is the synthetically-constructed oracle function (Definition 4.2.7)

$$\mathcal{O} := M(\frac{1}{3}, -1),$$

using the branching searcher.

ε	$((p_1)_{\delta_1}, \delta_1)$	$((p_2)_{\delta_2}, \delta_2)$	$\frac{(p_1)_{\delta_1}+1}{\delta_1}$	$\frac{(p_2)_{\delta_2}+1}{\delta_2}$	Time (s)
1	$(0, 2)$	$(-2048, 11)$	0.25	−0.999511719	
2	$(2, 3)$	$(-8192, 13)$	0.375	−0.99987793	2.38
3	$(2, 3)$	$(-32768, 15)$	0.375	−0.999969482	9.67
4	$(10, 5)$	$(-131072, 17)$	0.34375	−0.999998093	211.59
5	$(10, 5)$	$(-524288, 19)$	0.34375	−0.999998093	871.70

The graph below shows the oracle function $\mathcal{O}$ (in red) plotted against the function $M((p_1)^5, (p_2)^5)$ (in blue) where $(p_i)^5$ are the p_i computed when $\varepsilon := 3$, for the interval $[-1, 1]$



The efficiency is only mildly improved in this case. This example shows that we require better efficiency-minded approaches for search with multiple variables.

Chapter 7

Conclusion

7.1 Summary of contributions

This thesis has developed, in a constructive and univalent Agda formalisation (Chapter 2), a framework for performing search, optimisation and regression (Chapter 4) on a wide class of types given by closeness spaces and uniformly continuously searchable types (Chapter 3). Furthermore, we formally proved that uniformly continuously searchable types are closed under countable products (Theorem 3.3.14).

The Escardó-Simpson interval object (Section 5.1) and the ternary signed-digit encodings (Section 5.2) are formalised within our Agda library, and we verify the correctness of the operations on the latter using the former (Section 5.2.3).

We extracted examples of our formal framework directly from the Agda proofs, showing that we can perform search, optimisation and regression on formally verified functions of the ternary signed-digit encodings (Section 6.1.3).

We formalised the structure of another type for exact real computation, the ternary Boehm encodings (Section 5.3), and informally implemented search, optimisation and regression algorithms on this type in Java, in a way which reflects the formal Agda framework (Section 6.2.2).

Finally, we discussed the implementation of more efficient algorithms inspired by our formal approach, and gave some examples of their evaluation using Java (Section 6.2.3).

7.2 Further work

7.2.1 Verification of the order and closeness relations on ternary-signed digit encodings

Although we have formalised the correctness of the functions that we search, optimise and regress on the ternary signed-digit encodings $\mathfrak{3}^{\mathbb{N}}$ using the interval object $\mathbb{I}$ (Section 5.2.3), we have not formally proved that search, optimisation and regression on these representations amounts to those processes on the reals themselves.

In order to achieve this, we will need to verify that the *real-order preserving orders* (Definitions 6.1.7 and 6.1.9) do indeed preserve the numerical order on the interval object. However, this task may be somewhat involved, as the notion of an order on $\mathbb{I}$ is not well established. We therefore seek to establish this notion in our formalisation of the interval object, and then to verify our orders on $\mathfrak{3}^{\mathbb{N}}$.

Following this, we will also need to verify that the discrete-sequence closeness relation on $\mathfrak{3}^{\mathbb{N}}$ (Corollary 6.1.1) relates to a notion of a metric on the interval object $\mathbb{I}$, in the way informally described in Remark 3.2.20.

7.2.2 Verification of arithmetic on ternary Boehm encodings

We verified the arithmetic operations on the ternary signed-digit encodings, though we have not yet done this on the ternary Boehm encodings. In order to achieve this, we have begun to develop machinery for completing continuous functions approximated via dyadic interval codes $\mathbb{Z}^3$ into the equivalent function on ternary Boehm encodings $\mathbb{T}$, which automatically verifies it relative to the Dedekind reals $\mathbb{R}$. Using this machinery, the idea is that we can define a wide variety of operations by following the same blueprint each time.

We already utilise this machinery in our informal Java implementation, though the formal work still relies on conjectures that are fairly open and which require more work to prove both informally and formally. Although this this work is ongoing, we give the general idea in this section.

Remark 7.2.1. In this section, we use the notation $\{x_i\} := \{x_0, ..., x_{n-1}\}$ for an n-sized vector.

Definition 7.2.2. A multivariable function $f : \mathbb{R}^n \to \mathbb{R}$ is approximated by a *dyadic interval approximator* $A : ((\mathbb{Z}^3)^n \to \mathbb{Z}^3$ if,

1. Given two n-dimensional vectors of dyadic interval codes $\{(k_i, c_i, p_i)\}, \{(j_i, b_i, q_i)\} \colon (\mathbb{Z}^3)^n$ and one of dyadic rationals $\{w_i\} \colon \mathbb{Z}[1/2]^n$ such that $\frac{k_i}{2^{p_i}} \leq \frac{j_i}{2^{q_i}} \leq w_i \leq \frac{b_i}{2^{q_i}} \leq \frac{c_i}{2^{p_i}}$, it is the case that $\frac{Ak}{2^{Ap}} \leq \frac{Aj}{2^{Aq}} \leq f(w) \leq \frac{Ab}{2^{Aq}} \leq \frac{Ac}{2^{Ap}}$, where $(Ak, Ac, Ap) := A(\{(k_i, c_i, p_i)\})$ and $(Aj, Ab, Aq) := A(\{(j_i, b_i, q_i)\})$,
2. Given dyadic intervals $\{(a_i, b_i)\} \colon (\mathbb{Z}[1/2]^I)^n$ and required distance $\varepsilon \colon \mathbb{Z}[1/2]$, it is the case that there are distances $\{\delta_i\} \colon \mathbb{Z}[1/2]^n$ such that for all vectors $\{(k_i, c_i, p_i)\} \colon (\mathbb{Z}^3)^n$ satisfying $a_i \leq \frac{k_i}{2^{p_i}}, \frac{c_i}{2^{p_i}} \leq b_i$ and $\frac{c_i - k_i}{2^{p_i}} \leq \delta_i$, we have $\frac{Ac - Ak}{2^{Ap}} \leq \varepsilon$ where $(Ak, Ac, Ap) := A(\{(k_i, c_i, p_i)\})$.

The first condition determines that by refining our inputs we also refine the output, whereas the second determines that the approximator is uniformly continuous on any compact interval.

Example 7.2.3 (Addition's dyadic interval approximator)**.** Addition on dyadic-rational intervals is defined by:

$$\left[\frac{k_1}{2^{p_1}}, \frac{c_1}{2^{p_1}}\right] + \left[\frac{k_2}{2^{p_2}}, \frac{c_2}{2^{p_2}}\right] := \left[\frac{2^{p_2 - \min(p_1,p_2)} k_1 + 2^{p_1 - \min(p_1,p_2)} k_2}{2^{\max(p_1,p_2)}}, \frac{2^{p_2 - \min(p_1,p_2)} c_1 + 2^{p_1 - \min(p_1,p_2)} c_2}{2^{\max(p_1,p_2)}}\right],$$

Therefore, its dyadic interval approximator is defined:

$$A((k_1, c_1, p_1), (k_2, c_2, p_2)) := (2^{p_2 - \min(p_1,p_2)} k_1 + 2^{p_1 - \min(p_1,p_2)} k_2, 2^{p_2 - \min(p_1,p_2)} c_1 + 2^{p_1 - \min(p_1,p_2)} c_2, \max(p_1, p_2)).$$

We use the interval approximator to define the corresponding function on dyadic interval codes.

Definition 7.2.4. Given a multivariable function $f \colon \mathbb{R}^n \to \mathbb{R}$ which is approximated by a given dyadic interval approximator $A \colon (\mathbb{Z}^3)^n \to \mathbb{Z}^3$, the *corresponding function on dyadic interval codes* is defined as follows:

$$f' : (\mathbb{Z} \to \mathbb{Z}^3)^n \to (\mathbb{Z} \to \mathbb{Z}^3),$$
$$f'(\{\chi_i\})_n := A(\{(\chi_i)_n\}).$$

In order to encode a real number, the output of this corresponding function must be nested and positioned (recall these properties from Corollary 5.3.12).

Lemma 7.2.5. *Given a multivariable function $f\colon \mathbb{R}^n \to \mathbb{R}$ which is approximated by a given dyadic interval approximator $A\colon (\mathbb{Z}^3)^n \to \mathbb{Z}^3$, if the input sequences of dyadic interval codes $\{\chi_n\}\colon (\mathbb{Z} \to \mathbb{Z}^3)^n$ are nested then the output of the corresponding function on dyadic interval codes applied at these arguments $f'(\{\chi_i\})$ is nested.*

Proof. By the first condition.

Conjecture 7.2.6. *Given a multivariable function $f\colon \mathbb{R}^n \to \mathbb{R}$ which is approximated by a given dyadic interval approximator $A\colon (\mathbb{Z}^3)^n \to \mathbb{Z}^3$, if the input sequences of dyadic interval codes $\{\chi_n\}\colon (\mathbb{Z} \to \mathbb{Z}^3)^n$ are nested and positioned then the output of the corresponding function on dyadic interval codes applied at these arguments $f'(\{\chi_i\})$ is positioned.*

Using the above, we prove that the corresponding function on dyadic interval codes correctly realises the original function.

Conjecture 7.2.7. *Given a multivariable function $f\colon \mathbb{R}^n \to \mathbb{R}$ which is approximated by a given dyadic interval approximator $A\colon (\mathbb{Z}^3)^n \to \mathbb{Z}^3$, the corresponding function on dyadic interval codes $f'\colon (\mathbb{Z} \to \mathbb{Z}^3)^n \to (\mathbb{Z} \to \mathbb{Z}^3)$ is such that*

$$f(\{\llparenthesis \chi_i \rrparenthesis'\}) = \llparenthesis f'(\{\chi_i\}), \rrparenthesis'$$

for all $\{\chi_i\}\colon (\mathbb{Z} \to \mathbb{Z}^3)^n$ that are nested and positioned.

Proof. By Lemmas 7.2.5 and 5.3.10 and Conjecture 7.2.6.

Once we have approximated a function on dyadic interval codes, we next convert it into the equivalent operation on ternary interval codes. This is done using a special function $\mathsf{join}\colon (\mathbb{Z} \to \mathbb{Z}^3) \to (\mathbb{Z} \to \mathbb{Z}^2)$.

Conjecture 7.2.8. *There is a function $\mathsf{join}'\colon \mathbb{Z}^3 \to \mathbb{Z}^2$, which takes as input a dyadic interval code and outputs the narrowest ternary interval code that covers it.*

Definition 7.2.9. The function $\mathsf{join}\colon (\mathbb{Z} \to \mathbb{Z}^3) \to (\mathbb{Z} \to \mathbb{Z}^2)$, which converts a sequence of dyadic interval codes into a sequence of ternary interval codes, is defined as follows:

$$\mathsf{join} := \mathsf{map}(\mathsf{join}').$$

In order to encode a real number, this sequence of ternary interval codes must be nested and positioned.

Conjecture 7.2.10. *Given a sequence of dyadic interval codes $\chi \colon \mathbb{Z} \to \mathbb{Z}^3$, if χ is nested and positioned then so is* $\mathsf{join}(\chi)$.

Furthermore, joining the sequence does not change the real that is encoded.

Conjecture 7.2.11. *Given a sequence of dyadic interval codes $\chi \colon \mathbb{Z} \to \mathbb{Z}^3$, $(\!|\chi|\!)' = (\!|\mathsf{join}(\chi)|\!)'$.*

Definition 7.2.12. Given a multivariable function $f \colon \mathbb{R}^n \to \mathbb{R}$ which is approximated by a given dyadic interval approximator $A \colon (\mathbb{Z}^3)^n \to \mathbb{Z}^3$, the *corresponding function on ternary interval codes* is defined as follows:

$$\begin{aligned} f'' &: (\mathbb{Z} \to \mathbb{Z}^2)^n \to (\mathbb{Z} \to \mathbb{Z}^2), \\ f''(\{\chi_i\}) &:= \mathsf{join}(f'(\{\mathsf{map}(\text{to-dcode}, \chi_i)\})). \end{aligned}$$

Conjecture 7.2.13. *Given a multivariable function $f \colon \mathbb{R}^n \to \mathbb{R}$ which is approximated by a given dyadic interval approximator $A \colon (\mathbb{Z}^3)^n \to \mathbb{Z}^3$, the corresponding function on ternary interval codes $f'' \colon (\mathbb{Z} \to \mathbb{Z}^2)^n \to (\mathbb{Z} \to \mathbb{Z}^2)$ is such that,*

$$f(\{(\!|\chi_i|\!)''\}) = (\!|f''(\{\chi_i\})|\!)''$$

for all $\{\chi_i\} \colon (\mathbb{Z} \to \mathbb{Z}^2)^n$ that are nested and positioned.

Proof. By Lemma 5.3.10 and Conjectures 7.2.7, 7.2.8, 7.2.10 and 7.2.11.

The final step is to normalise the inputs and output of this function.

Conjecture 7.2.14. *There is a function* $\mathsf{normalise} \colon (\mathbb{Z} \to \mathbb{Z}^2) \to (\mathbb{Z} \to \mathbb{Z}^2)$, *which takes a nested and positioned sequence of ternary interval codes and gives back a normalised sequence of ternary interval codes that represents the same real number.*

Definition 7.2.15. Given a multivariable function $f \colon \mathbb{R}^n \to \mathbb{R}$ which is approximated by a given dyadic interval approximator $A \colon (\mathbb{Z}^3)^n \to \mathbb{Z}^3$, the *corresponding function* *on ternary Boehm encodings* is defined as follows:

$$\begin{aligned}&\overline{f} : \mathbb{T}^n \to \mathbb{T},\\ &\overline{f}(\{\chi_i\}) := \mathsf{normalise}(f''(\{\mathsf{to\text{-}interval\text{-}seq}(\chi_i)\})),\end{aligned}$$

where $\mathsf{to\text{-}interval\text{-}seq}\colon \mathbb{T} \to (\mathbb{Z} \to \mathbb{Z}^2)$ is the map resulting from the equivalence between the ternary Boehm encodings and normalised sequences of ternary interval codes (Lemma 5.3.19).

Conjecture 7.2.16. *Given a multivariable function $f\colon \mathbb{R}^n \to \mathbb{R}$ which is approximated by a given dyadic interval approximator $A\colon (\mathbb{Z}^3)^n \to \mathbb{Z}^3$, the corresponding function on ternary Boehm encodings $\overline{f}\colon \mathbb{T}^n \to \mathbb{T}$ is such that,*

$$f(\{[\![\chi_i]\!]\}) = [\![\overline{f}(\{\chi_i\})]\!]$$

for all $\{\chi_i\}\colon \mathbb{T}^n$.

Proof. By Corollary 5.3.12 and Conjectures 7.2.13 and 7.2.14

This machinery amounts to showing that, for the above definitions, the following diagram commutes:

$$\begin{tikzcd}
\mathbb{T}^n \arrow[r, shift left] \arrow[d, dashed, "\overline{f}"'] \arrow[rrr, dashed, bend left, "{[\![-]\!]^n}"] & (\mathbb{Z} \to \mathbb{Z}^2)^n \arrow[l, shift left, "\mathsf{normalise}"] \arrow[r, shift left] \arrow[d, dashed, "\overline{f}''"] & (\mathbb{Z} \to \mathbb{Z}^3)^n \arrow[l, shift left, "\mathsf{join}"] \arrow[r] \arrow[d, "\overline{f}'"] & \mathbb{R}^n \arrow[d, "f"] \\
\mathbb{T} \arrow[r, shift left] \arrow[rrr, dashed, bend right, "{[\![-]\!]}"'] & (\mathbb{Z} \to \mathbb{Z}^2) \arrow[l, shift left] \arrow[r, shift left] & (\mathbb{Z} \to \mathbb{Z}^3) \arrow[l, shift left] \arrow[r] & \mathbb{R}
\end{tikzcd}$$

7.2.3 Towards practical implementations of exact real search

Our further work is largely concerned with further formalisation and verification results, but there exists a loftier goal of this work: more efficient practical implementations that do not sacrifice the verified correctness.

In our AGDA formalisation, we have only used inefficient exhaustive methods for search, optimisation and regression. In Section 6.2.3, we discussed more efficient exhaustive methods via branch-and-bound techniques, and gave examples of our implementation in our Java code. The first step towards bridging the gap between correctness and efficiency in our work is the formalisation of the correctness of these methods — i.e., the development of general convergence theorems for branch-and-bound techniques in our Agda framework.

Following this, we would like to look at the development of efficient methods that utilise heuristics, and the development of efficient methods for the search, optimisation and regression of multivariable functions.

Appendix A

Formal Agda Framework

This thesis' primary contribution is that the lion's share of its *other* contributions are fully-formalised in the programming language and proof assistant Agda.

The formalisation is available for viewing on this thesis' GitHub repository. We outline the formalisation's files (divided into seven directories) below.

Chapter2

Finite.lagda.md contains the functions we require for finite linearly ordered types.

Vectors.lagda.md contains some additional functions we require for vectors.

Sequences.lagda.md contains the functions we require for sequences.

Chapter3

ClosenessSpaces.lagda.md contains the formalisation of closeness spaces and their related lemmas, as described in Section 3.2.

ClosenessSpaces-Examples.lagda.md contains the examples of closeness spaces, as described in Section 3.2.5.

SearchableTypes.lagda.md contains the formalisation of searchable and uniformly continuously searchable types, as described in Sections 3.1 and 3.3.

SearchableTypes-Examples.lagda.md contains the examples of uniformly continuously searchable types, as described in Section 3.3.2. The formalised Tychonoff theorem for uniformly continuously searchable types (Theorem 3.3.14) is at the bottom of this file.

PredicateEquality.lagda.md contains a small number of lemmas for proving the equality of uniformly continuous and decidable predicates. These lemmas are used in the formalisations of Lemma 3.3.9 and Theorem 3.3.14.

Chapter4

ApproxOrder.lagda.md contains the formalisation of approximate linear preorders, as described in Section 4.1.1.

ApproxOrder-Examples.lagda.md contains the examples of approximate linear preorders, as described in Section 4.1.1.

GlobalOptimisation.lagda.md contains the formalisation of our generalised, type-theoretic variant of global optimisation, as described in Section 4.1.2. The global optimisation algorithm Theorem 4.1.26 is at the bottom of this file.

ParametricRegression.lagda.md contains the formalisation of our generalised, type-theoretic variant of parametric regression, as described in Section 4.2. The parametric regression convergence theorems Theorems 4.2.6, 4.2.9 and 4.2.10 are at the bottom of this file.

Chapter5

IntervalObject.lagda.md contains the formalisation of the Escardó-Simpson interval object, as described in Section 5.1.

IntervalObjectApproximation.lagda.md contains the formal verification of finite approximations for the interval object, as described in Section 5.1.

SignedDigit.lagda.md contains the formalisation of the ternary signed-digit encodings and their arithmetic, as described in Section 5.2.

SignedDigitIntervalObject.lagda.md contains the formal verification of the ternary signed-digit encodings using the interval object, as described in Section 5.2.

BoehmVerification.lagda.md contains our current formalised work on the ternary Boehm encodings, as described in Section 5.3.

BelowAndAbove.lagda.md contains a variety of lemmas concerning the structure of the ternary Boehm encodings.

Chapter6

SequenceContinuity.lagda.md contains the definitions and proofs concerning the specialised form of uniform continuity for sequence functions, as seen in Section 6.1.1.

SignedDigitSearch.lagda.md contains the corollaries required to instantiate our framework for search, optimisation and regression on the ternary signed-digit encodings, as described in Section 6.1.1.

SignedDigitOrder.lagda.md contains the formalisation of the real-order preserving orders, which allows us to correctly order the reals using the ternary signed-digit encodings, as seen in Section 6.1.1.

SignedDigitContinuity.lagda.md contains the proofs that the functions we have defined for exact real arithmetic on the ternary signed-digit encodings are uniformly continuous, as described in Section 6.1.1.

SignedDigitExamples.lagda.md contains the examples of our formal framework for search, optimisation and regression applied to the ternary signed-digits, as described in Section 6.1.3.

Main.lagda.md is a file that can be compiled into a Haskell file in order to run the examples, should the reader desire. Before compiling, ensure that the example being computed (from `SignedDigitExamples.lagda.md`) is the one desired at the correct level of precision; then, run `agda --compile TWA/Thesis/Chapter6/Main.lagda.md` in the `source` folder of the branch. The code can then be ran by performing `ghci MAlonzo/Code/TWA/Thesis/Chapter6/Main.hs`. Once `ghci` has loaded, type `main` and hit enter to run the example.

Appendix B

Java Implementation of Ternary Boehm Encodings

The examples of search, optimisation and regression performed on ternary Boehm encodings given in Section 6.2 are implemented in a small Java library written by Andrew Sneap and Todd Waugh Ambridge.

Note that we do not utilise any of Boehm's code (from [Boe99]), instead re-implementing both the representation — due to our modifications (detailed in Section 5.3) — and the basic arithmetic functions — in order to align with the machinery detailed in Section 7.2.2, which allows the function's modulus of continuity to be easily extracted.

The implementation is available for viewing on this thesis' GitHub repository. We outline the implementation's files (divided into seven packages) below, using the type-theoretic parlance of the rest of the thesis.

DyadicsAndIntervals

Dyadic.java implements dyadic rational numbers $\mathbb{Z}[{}^1\!/\!{}_2]$ as pairs $\mathbb{Z}\times\mathbb{Z}$, where $(k, i) : \mathbb{Z}\times\mathbb{Z}$ represents the dyadic $\frac{k}{2^i}$. The structural operations defined in Section 5.3 are extended here to the dyadics. Arithmetic and comparison operations on dyadics are also defined here.

DyadicIntervalCode.java implements dyadic interval codes $\mathbb{Z}^3$ (Definition 5.3.11), where $(k, c, p) : \mathbb{Z}^3$ represents the dyadic interval $[\frac{k}{2^p}, \frac{c}{2^p}]$. We can yield the dyadic endpoints of a dyadic interval code. Again, we extend the structural operations defined in Section 5.3

to these interval codes. Arithmetic and comparison operations on dyadic interval codes are also defined here.

TernaryIntervalCode.java implements ternary interval codes $\mathbb{Z}^2$ (Definition 5.3.13), where $(k, p) \colon \mathbb{Z}^3$ represents the dyadic interval $[\frac{k}{2^p}, \frac{k+2}{2^p}]$. We can convert any ternary interval code into a dyadic interval code. Again, we extend the structural operations defined in Section 5.3 to these interval codes. Comparison operations on ternary interval codes are also defined here. Functions which 'discretise' the interval (i.e. convert it into the intervals directly below it on a higher-precision level) are implemented here.

TernaryBoehm

TBEncoding.java implements ternary Boehm encodings $\mathbb{T}$ (Definition 5.3.3). We can convert dyadics, integers and ternary interval codes into ternary Boehm encodings. A ternary Boehm encoding can be converted into a sequence of dyadic interval codes or ternary interval codes. Arithmetic on the ternary Boehm encodings is defined by the application of particular `CFunction` objects in this file.

FunctionsAndPredicates

CFunction.java implements multivariable continuous functions $f \colon \mathbb{T}^n \to \mathbb{T}$ on the ternary Boehm encodings. A function is constructed by giving its interval approximator $(\mathbb{Z}^3)^n \to \mathbb{Z}^3$ on ternary interval codes, as well as information about the continuity of that function. This interval approximated is 'completed' to a function on the ternary Boehm encodings, in the manner described in Section 7.2.2. This class contains a large number of static methods that define functions such as negation, addition and multiplication. There are also a variety of methods for the composition of functions, which automatically computes the new interval approximators and continuity information.

UCUnaryPredicate.java implements uniformly continuous unary predicates $p \colon \mathbb{T} \to \Omega$. A predicate is constructed by giving a function `TBEncoding -> Boolean` and by giving the predicate's modulus of uniform continuity. There are also static methods for some simple predicates, such as $p(x) := x \leq^{\epsilon} y$ for a given $y \colon \mathbb{T}$. Importantly, there is a constructor for building predicates $p(x) := p'(f(x))$ defined by functions f — this automatically works out the modulus of uniform continuity of the resulting predicate by the moduli of uniform continuity of the underlying function and predicate.

UCBinaryPredicate.java implements uniformly continuous binary predicates.

Search

SearchUnary.java implements a uniformly continuous search algorithm for unary predicates. The constructor takes a unary predicate and a ternary interval code to search for an answer, while the function `search()` actually performs the algorithm. The algorithm sets the bounds on the search candidates on the precision-level required (i.e. the level given by the modulus of uniform continuity) and then tests each candidate in numerical order. As soon as an answer is found, it is returned by the algorithm, and if no answer is found then the algorithm exhausts the space and states that no answer was found.

SearchBinary.java implements the uniformly continuous search algorithm for binary predicates.

Optimisation

Optimisation.java implements the global optimisation procedure (Theorem 4.1.26) for unary functions $f\colon \mathbb{T} \to \mathbb{T}$. The constructor takes a function, a requested output precision $\epsilon\colon \mathbb{Z}$ and a ternary interval code to search for an ϵ-global minimum. The input precision δ required to achieve the requested output precision is computed and the δ-net of search candidates is generated. Then, each candidate is checked in numerical order until the net is completely exhausted. The algorithm keeps track of the current ϵ-minimum argument to the function. Once the net is exhausted, it returns this ϵ-minimum argument.

OptimisationHeuristic.java implements a branch-and-bound (Section 6.2.3) global optimisation procedure for unary functions $f\colon \mathbb{T} \to \mathbb{T}$. This differs to the usual algorithm in that the a δ-net is not computed in advance of the optimisation. Instead, the initial candidate ternary interval code is branched into the two ternary interval codes below it, and their output bounds for the function are computed using the interval approximator and continuity information held in the `CFunction` object. This process repeats, and any candidate interval that has a lower output bound greater than another candidate interval's upper output bound is discarded, as this candidate can clearly not contain a minimum.

Regression

Regression.java implements regression algorithms (Section 4.2) via the above optimisation and search algorithms. The construction of a regression algorithm requires an oracle function, model function and list of predictor observations. The loss function used is defined as `averageModelOracleDistance()`, which sums the distance between the two functions' outcomes on the observations.

Examples

Examples.java contains the examples of search, optimisation and regression that we described in Section 6.2.2.

Bibliography

[AM00] Götz Alefeld and Günter Mayer. "Interval Analysis: Theory and Applications". In: *Journal of Computational and Applied Mathematics* 121.1 (2000), pp. 421–464. ISSN: 0377-0427. DOI: https://doi.org/10.1016/S0377-0427(00)00342-3 (cit. on p. 1).

[Amb20a] Todd Waugh Ambridge. *EscardoSimpson-LICS2001*. TypeTopology. 2020. URL: https://www.cs.bham.ac.uk/~mhe/TypeTopology/TWA.Escardo-Simpson-LICS2001.html (cit. on p. 89).

[Amb20b] Todd Waugh Ambridge. *Formalising the Escardó-Simpson Closed Interval Axiomatisation in Univalent Type Theory*. Talk given at the Workshop on Homotopy Type Theory/ Univalent Foundations 2020. 2020. URL: https://hott-uf.github.io/2020/HoTTUF_2020_paper_17.pdf (cit. on p. 89).

[Amb20c] Todd Waugh Ambridge. *SIP-IntervalObject*. TypeTopology. 2020. URL: https://www.cs.bham.ac.uk/~mhe/TypeTopology/TWA.SIP-IntervalObject.html (cit. on p. 98).

[ANST20] Benedikt Ahrens, Paige Randall North, Michael Shulman, and Dimitris Tsementzis. "A Higher Structure Identity Principle". In: *Proceedings of the 35th Annual ACM/IEEE Symposium on Logic in Computer Science*. LICS '20. Saarbrücken, Germany: Association for Computing Machinery, 2020, pp. 53–66. ISBN: 9781450371049. DOI: 10.1145/3373718.3394755. URL: https://doi.org/10.1145/3373718.3394755 (cit. on p. 98).

[ANST21] Benedikt Ahrens, Paige Randall North, Michael Shulman, and Dimitris Tsementzis. "The univalence principle". In: *arXiv preprint arXiv:2102.06275* (2021) (cit. on p. 21).

[Bau08] Andrej Bauer. “Efficient computation with Dedekind reals”. In: *Fifth International Conference on Computability and Complexity in Analysis, Hagen, Germany*. Citeseer. 2008 (cit. on p. 89).

[BB12] Errett Bishop and Douglas Bridges. *Constructive Analysis*. Vol. 279. Springer-Verlag, 2012 (cit. on p. 2).

[BCDE23] Marc Bezem, Thierry Coquand, Peter Dybjer, and Martín Hötzel Escardó. “Type Theory with Explicit Universe Polymorphism”. In: *28th International Conference on Types for Proofs and Programs (TYPES 2022)*. Vol. 269. Leibniz International Proceedings in Informatics (LIPIcs). Dagstuhl, Germany: Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2023, 13:1–13:16. ISBN: 978-3-95977-285-3 (cit. on p. 7).

[BCRO86] Hans-Juergen Boehm, Robert Cartwright, Mark Riggle, and Michael J. O’Donnell. “Exact Real Arithmetic: A Case Study in Higher Order Programming”. In: *Proceedings of the 1986 ACM Conference on LISP and Functional Programming*. LFP 86. Cambridge, Massachusetts, USA: ACM, 1986, pp. 162–173. DOI: 10.1145/319838.319860 (cit. on pp. 1, 103).

[BDN09] Ana Bove, Peter Dybjer, and Ulf Norell. “A Brief Overview of Agda – A Functional Language with Dependent Types”. In: *Theorem Proving in Higher Order Logics: 22nd International Conference Proceedings 22*. 2009, pp. 73–78 (cit. on pp. 2, 5).

[Ber09] Ulrich Berger. “From Coinductive Proofs to Exact Real Arithmetic”. In: *Computer Science Logic: 23rd international Workshop, 18th Annual Conference of the EACSL Proceedings 23*. Springer. 2009, pp. 132–146 (cit. on pp. 2, 104).

[Ber90] Ulrich Berger. “Totale Objekte und Mengen in der Bereichstheorie”. PhD thesis. Uitgever niet vastgesteld, 1990 (cit. on pp. 2, 31).

[BM07] Stephen Boyd and Jacob Mattingley. “Branch and Bound Methods”. In: *Lecture Notes (Contrained Optimization II), Stanford University* 2006 (2007), p. 07 (cit. on p. 162).

[Boe17] Hans-Juergen Boehm. “Small-Data Computing: Correct Calculator Arithmetic”. In: *Communications of the ACM* 60.8 (2017), pp. 44–49. DOI: 10.1145/2911981 (cit. on pp. 1, 117).

[Boe20] Hans-Juergen Boehm. "Towards an API for the Real Numbers". In: *Proceedings of the 41st ACM SIGPLAN International Conference on Programming Language Design and Implementation, PLDI.* ACM, 2020, pp. 562–576 (cit. on pp. 3, 88, 117, 125).

[Boe99] Hans-Juergen Boehm. *Constructive Reals Calculator.* 1999. URL: https://hboehm.info/new_crcalc/CRCalc.html (cit. on pp. 3, 117, 178).

[Boo20] Auke Bart Booij. "Analysis in Univalent Type Theory". PhD thesis. University of Birmingham, 2020 (cit. on p. 88).

[BPRS18] Atilim Gunes Baydin, Barak A Pearlmutter, Alexey Andreyevich Radul, and Jeffrey Mark Siskind. "Automatic Differentiation in Machine Learning: A Survey". In: *Journal of Marchine Learning Research* 18 (2018), pp. 1–43 (cit. on p. 70).

[CD06] Alberto Ciaffaglione and Pietro Di Gianantonio. "A Certified, Corecursive Implementation of Exact Real Numbers". In: *Theoretical Computer Science* 351.1 (2006), pp. 39–51 (cit. on p. 2).

[Chu40] Alonzo Church. "A Formulation of the Simple Theory of Types". In: *The Journal of Symbolic Logic* 5.2 (1940), pp. 56–68 (cit. on p. 6).

[Cla99] Jens Clausen. "Branch and bound algorithms - principles and examples". In: *Department of Computer Science, University of Copenhagen* (1999), pp. 1–30 (cit. on p. 162).

[CM02] Vladimir Cherkassky and Yunqian Ma. "Selecting of the Loss Function for Robust Linear Regression". In: *Neural computation* (2002) (cit. on p. 69).

[Di 93] Pietro Di Gianantonio. "A Functional Approach to Computability on Real Numbers". In: *Bulletin-European Association For Theoretical Computer Science* 50 (1993). URL: https://users.dimi.uniud.it/~pietro.digianantonio/papers/ (cit. on pp. 2, 88, 104).

[dJon23] Tom de Jong. *Domain Theory in Constructive and Predicative Univalent Foundations.* 2023. arXiv: 2301.12405 [cs.LO] (cit. on p. 5).

[ES01] Martín Hötzel Escardó and Alex K Simpson. "A Universal Characterization of the Closed Euclidean Interval". In: *Proceedings 16th Annual IEEE Symposium on Logic in Computer Science.* IEEE. 2001, pp. 115–125. DOI: 10.1109/LICS.2001.932488 (cit. on pp. 2, 89, 94, 97).

[ES16] Martín Hötzel Escardó and Thomas Streicher. "The intrinsic topology of Martin-Löf universes". In: *Annals of Pure and Applied Logic* 167.9 (2016). Fourth Workshop on Formal Topology (4WFTop), pp. 794–805. ISSN: 0168-0072. DOI: https://doi.org/10.1016/j.apal.2016.04.010. URL: https://www.sciencedirect.com/science/article/pii/S0168007216300409 (cit. on p. 74).

[Esc08] Martín Hötzel Escardó. "Exhaustible Sets in Higher-Type Computation". In: *Logical methods in computer science* 4 (2008) (cit. on pp. 2, 31, 36, 65).

[Esc11a] Martín Hötzel Escardó. *CompactTypes.* TYPETOPOLOGY. 2011. URL: https://www.cs.bham.ac.uk/~mhe/TypeTopology/TypeTopology.CompactTypes.html (cit. on pp. 2, 31).

[Esc11b] Martín Hötzel Escardó. *Real Number Computation in Haskell with Real Numbers Represented as Infinite Sequences of Digits.* 2011. URL: https://www.cs.bham.ac.uk/~mhe/papers/fun2011.lhs (cit. on pp. 2, 31, 104, 108, 113, 115, 145).

[Esc12] Martín Hötzel Escardó. *The Topology of Seemingly Impossible Functional Programs.* POPL TutorialFest. 2012. URL: https://www.cs.bham.ac.uk/~mhe/.talks/popl2012/escardo-popl2012.pdf (cit. on p. 31).

[Esc13a] Martín Hötzel Escardó. "Algorithmic Solution of Higher Type Equations". In: *Journral of Logic and Computation* 23.4 (2013), pp. 839–854. DOI: 10.1093/logcom/exr048 (cit. on p. 31).

[Esc13b] Martín Hötzel Escardó. *Universes.* TYPETOPOLOGY. 2013. URL: https://www.cs.bham.ac.uk/~mhe/TypeTopology/MLTT.Universes.html (cit. on p. 8).

[Esc19] Martín Hötzel Escardó. "Introduction to Univalent Foundations of Mathematics with Agda".

In: *CoRR* abs/1911.00580 (2019). arXiv: 1911.00580.
URL: http://arxiv.org/abs/1911.00580 (cit. on pp. 8, 13).

[Esc20] Martín Hötzel Escardó. *SIP*. TypeTopology. 2020.
URL: https://www.cs.bham.ac.uk/~mhe/TypeTopology/UF.SIP.html
(cit. on p. 98).

[Esc21] Martín Hötzel Escardó. *Kuratowski*. TypeTopology. 2021. URL:
https://www.cs.bham.ac.uk/~mhe/agda-new/Fin.Kuratowski.html
(cit. on p. 26).

[Esc23] et al. Escardó Martín Hötzel. *TypeTopology. Various New Theorems in Univalent Mathematics Written in Agda*. 2023.
URL: https://www.cs.bham.ac.uk/~mhe/TypeTopology/
(cit. on pp. 2, 4–5).

[Esc98] Martín Hötzel Escardó. “Effective and Sequential Definition by Cases on the Reals via Infinite Signed-Digit Numerals”.
In: *Electronic Notes in Theoretical Computer Science* 13 (1998), pp. 53–68
(cit. on p. 135).

[FG09] Christodoulos A Floudas and Chrysanthos E Gounaris.
“A review of Recent Advances in Global Optimization”.
In: *Journal of Global Optimization* 45 (2009), pp. 3–38 (cit. on p. 81).

[GA21] Dan R Ghica and Todd Waugh Ambridge.
“Global Optimisation with Constructive Reals”. In: *2021 36th Annual ACM/IEEE Symposium on Logic in Computer Science (LICS)*. IEEE. 2021, pp. 1–13 (cit. on pp. 82, 89, 133).

[Gam09] Nicola Gambino. *Lectures on Dependent Type Theory*. 2009.
URL: https://www.cs.le.ac.uk/events/mgs2009/courses/gambino/lecturenotes-gambino.pdf (cit. on p. 7).

[Gam11] Nicola Gambino. “The Univalence Axiom and Function Extensionality”.
In: *The Oberwolfach Mini-Workshop on the Homotopy Interpretation of Constructive Type Theory* (2011). Notes taken by Chris Kapulkin and Peter LeFanu Lumsdaine. URL: https://www.math.uwo.ca/faculty/kapulkin/notes/ua_implies_fe.pdf
(cit. on p. 22).

[GNSW07] Herman Guevers, Milad Niqui, Bas Spitters, and Freek Wiedijk. "Constructive analysis, types and exact real numbers". In: *Mathematical Structures in Computer Science* 17.1 (2007), pp. 3–36. DOI: 10.1017/S0960129506005834 (cit. on p. 88).

[Gol91] David Goldberg. "What Every Computer Scientist Should Know about Floating-Point Arithmetic". In: *ACM computing surveys (CSUR)* 23.1 (1991), pp. 5–48 (cit. on p. 1).

[Imp15] Agda Implementors. *Universe Levels*. AGDA documentation. 2015. URL: https://agda.readthedocs.io/en/latest/language/universe-levels.html (cit. on p. 8).

[JR21] Kai Jia and Martin Rinard. "Exploiting Verified Neural Networks via Floating Point Numerical Error". In: *Static Analysis: 28th International Symposium Proceedings*. Berlin, Heidelberg: Springer-Verlag, 2021, pp. 191–205. ISBN: 978-3-030-88805-3. DOI: 10.1007/978-3-030-88806-0_9. URL: https://doi.org/10.1007/978-3-030-88806-0_9 (cit. on p. 1).

[Kap01] Irving Kaplansky. *Set Theory and Metric Spaces*. Allyn and Bacon Series in Advanced Mathematics. Boston: AMS Chelsea Publishing, 2001 (cit. on p. 38).

[Kea92] R. B. Kearfott. "An Interval Branch and Bound Algorithm for Bound Constrained Optimization Problems". In: *Journal of Global Optimization* 2 (1992), pp. 259–280. URL: https://doi.org/10.1007/BF00171829 (cit. on pp. 71, 163).

[KK11] Karin Usadi Katz and Mikhail G Katz. "Meaning in Classical Mathematics: Is it at Odds with Intuitionism?" In: *arXiv preprint arXiv:1110.5456* (2011) (cit. on p. 71).

[Lib08] Leo Liberti. "Introduction to Global Optimization". In: *Ecole Polytechnique* (2008) (cit. on pp. 1–2).

[Mar75] Per Martin-Löf. "An Intuitionistic Theory of Types: Predicative Part". In: *Studies in Logic and the Foundations of Mathematics*. Vol. 80. Elsevier, 1975, pp. 73–118. DOI: https://doi.org/10.1016/S0049-237X(08)71945-1 (cit. on pp. 5–7).

[MS84] Per Martin-Löf and Giovanni Sambin. *Intuitionistic Type Theory*. Vol. 9. Bibliopolis Naples, 1984.
URL: https://archive-pml.github.io/martin-lof/pdfs/Bibliopolis-Book-retypeset-1984.pdf (cit. on pp. 7, 13).

[NPS90] Bengt Nordström, Kent Petersson, and Jan M Smith. *Programming in Martin-Löf's Type Theory*. Vol. 200. Oxford University Press, Oxford, 1990 (cit. on pp. 2, 37).

[NTvD22] Lam M Nguyen, Trang H Tran, and Marten van Dijk. "New Perspective on the Global Convergence of Finite-Sum Optimization". In: (2022). URL: http://arxiv.org/licenses/nonexclusive-distrib/1.0/ (cit. on p. 70).

[Pie02] Benjamin C Pierce. *Types and Programming Languages*. MIT Press, 2002 (cit. on p. 6).

[Pin00] Allan Pinkus. "Weierstrass and Approximation Theory". In: *Journal of Approximation Theory* 107.1 (2000), pp. 1–66. DOI: https://doi.org/10.1006/jath.2000.3508 (cit. on pp. 84–85).

[Piy72] SA Piyavskii. "An Algorithm for Finding the Absolute Extremum of a Function". In: *USSR Computational Mathematics and Mathematical Physics* 12.4 (1972), pp. 57–67. URL: https://doi.org/10.1016/0041-5553(72)90115-2 (cit. on p. 71).

[Plu98] Dave Plume. *A Calculator for Exact Real Number Computation*. 1998 (cit. on pp. 2, 104).

[Rud64] Walter Rudin. *Principles of Mathematical Analysis*. Vol. 3. McGraw-hill New York, 1964 (cit. on p. 38).

[Scr07] Adam Scriven. *Functional algorithms for Exact Real Integration over Invariant Measures*. Master's thesis, Department of Computer Science, University of Birmingham. 2007 (cit. on p. 113).

[Shu12] Mike Shulman. *Propositions as Some Types and Algebraic Nonalgebraicity*. https://golem.ph.utexas.edu/category/2012/01/propositions_as_some_types_and.html. 2012 (cit. on p. 16).

[Shu18] Michael Shulman.
"Brouwer's fixed-point theorem in real-cohesive homotopy type theory".
In: *Mathematical Structures in Computer Science* 28.6 (2018), pp. 856–941.
DOI: 10.1017/S0960129517000147 (cit. on pp. 71, 89).

[Sim98] Alex K Simpson.
"Lazy Functional Algorithms for Exact Real Functionals".
In: *Mathematical Foundations of Computer Science: 23rd International Symposium. Proceedings 23.* Springer. 1998, pp. 456–464
(cit. on pp. 1–2, 31, 71).

[Sne21] Andrew Sneap. *DedekindReals.* TypeTopology. 2021. URL: https://www.cs.bham.ac.uk/~mhe/TypeTopology/DedekindReals.index.html
(cit. on p. 121).

[Str93] Thomas Streicher. "Investigations into intensional type theory".
PhD thesis. 1993 (cit. on p. 13).

[Sut09] W. A. (Wilson Alexander) Sutherland.
Introduction to metric and topological spaces / Wilson A Sutherland. eng.
2009 (cit. on p. 46).

[TD88] Anne Sjerp Troelstra and D. Dalen.
Constructivism in Mathematics: An Introduction.
Studies in logic and the foundations of mathematics ; v. 121, 123 v. 1. 1988.
URL: https://books.google.co.uk/books?id=EubuAAAAMAAJ
(cit. on pp. 35, 71, 81).

[Tre13] William F Trench. *Introduction to Real Analysis.* eng.
Place of publication not identified: A.T. Still University, 2013.
ISBN: 0130457868 (cit. on pp. 89, 91).

[Tur37] Alan Mathison Turing. "On computable Numbers, with an Application to the Entscheidungs Problem".
In: *Proceedings of the London Mathematical Society* 2.1 (1937), pp. 230–265
(cit. on p. 1).

[Uni13] The Univalent Foundations Program.
Homotopy Type Theory: Univalent Foundations of Mathematics. Institute for Advanced Study: https://homotopytypetheory.org/book, 2013
(cit. on pp. 5–7, 20, 25, 88).

[VAG+] Vladimir Voevodsky, Benedikt Ahrens, Daniel Grayson, et al. *UniMath — a computer-checked library of univalent mathematics.* available at `http://unimath.org`. DOI: `10.5281/zenodo.8427604`. URL: `https://doi.org/10.5281/zenodo.8427604` (cit. on p. 6).

[WCB+18] Naigang Wang, Jungwook Choi, Daniel Brand, Chia-Yu Chen, and Kailash Gopalakrishnan. "Training Deep Neural Networks with 8-bit Floating Point Numbers". In: *Advances in Neural Information Processing Systems.* Ed. by S. Bengio, H. Wallach, H. Larochelle, K. Grauman, N. Cesa-Bianchi, and R. Garnett. Vol. 31. Curran Associates, Inc., 2018. URL: `https://proceedings.neurips.cc/paper_files/paper/2018/file/335d3d1cd7ef05ec77714a215134914c-Paper.pdf` (cit. on p. 1).

[WK19] Shibo Wang and Pankaj Kanwar. *BFloat16: The secret to high performance on Cloud TPUs.* 2019. URL: `https://cloud.google.com/blog/products/ai-machine-learning/bfloat16-the-secret-to-high-performance-on-cloud-tpus` (cit. on p. 1).

[WR27] Alfred North Whitehead and Bertrand Arthur William Russell. *Principia Mathematica; 2nd ed.* Cambridge: Cambridge Univ. Press, 1927. URL: `https://cds.cern.ch/record/268025` (cit. on p. 6).

[YS09] Xin Yan and Xiaogang Su. *Linear Regression Analysis: Theory and Computing.* World Scientific. 2009 (cit. on pp. 1–2, 60, 82).