Three Equivalent Ordinal Notation Systems in Cubical Agda

Fredrik Nordvall Forsberg
Computer and Information Sciences
University of Strathclyde
Glasgow, United Kingdom
fredrik.nordvall-forsberg@strath.ac.uk

Chuangjie Xu
Mathematisches Institut
Ludwig-Maximilians-Universität München
Munich, Germany
cj-xu@outlook.com

Neil Ghani
Computer and Information Sciences
University of Strathclyde
Glasgow, United Kingdom
neil.ghani@strath.ac.uk

Abstract

We present three ordinal notation systems representing ordinals below ε_0 in type theory, using recent type-theoretical innovations such as mutual inductive-inductive definitions and higher inductive types. We show how ordinal arithmetic can be developed for these systems, and how they admit a transfinite induction principle. We prove that all three notation systems are equivalent, so that we can transport results between them using the univalence principle. All our constructions have been implemented in cubical Agda.

CCS Concepts • **Theory of computation → Proof theory**; **Type theory**.

Keywords Ordinal notation, Cantor normal form, inductive-inductive definitions, higher inductive types, cubical Agda.

ACM Reference Format:
Fredrik Nordvall Forsberg, Chuangjie Xu, and Neil Ghani. 2020. Three Equivalent Ordinal Notation Systems in Cubical Agda. In *Proceedings of the 9th ACM SIGPLAN International Conference on Certified Programs and Proofs (CPP '20), January 20–21, 2020, New Orleans, LA, USA.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3372885.3373835

1 Introduction

Ordinals and ordinal notation systems play an important role in program verification, since they can be used to prove termination of programs — using ordinals to verify that programs terminate was suggested already by Turing [29]. The idea is to assign an ordinal to each input, and then prove that the assigned ordinal decreases for each recursive call. Hence the program must terminate by the well-foundedness of the order on ordinals. At first, such proofs were carried out using pen and paper [11, 14], but with advances in proof assistants, also machine-checked proofs can be produced [21, 24]. As a first step, one must then represent ordinals inside a theorem prover. This is usually done via some kind of ordinal notation system (however see Blanchette *et al.* [4] for well-orders encoded directly in Isabelle/HOL, and Schmitt [24] for an axiomatic method, which is implemented in the KeY program verification system). Typically, ordinals are represented by trees [10, 12]; for instance, binary trees can represent the ordinals below ε_0 as follows: the leaf represents 0, and a tree with subtrees representing ordinals α and β represents the sum $\omega^\alpha + \beta$. However, an ordinal may have multiple such representations. As a result, traditional approaches to ordinal notation systems [5, 25, 27] usually have to single out a subset of ordinal terms in order to provide unique representations. In this paper, we show how modern type-theoretic features can be used to directly give faithful representations of ordinals below ε_0.

The first feature we use is mutual inductive-inductive definitions [22], which are well supported in the proof assistant Agda. This allows us to define an ordinal notation system for ordinals below ε_0, simultaneously with an order relation on it (Section 3.2). This means that we can recover uniqueness of representation, by insisting that subtrees representing ordinals are given in a decreasing order. This is similar to the traditional approach which first freely generate ordinal terms, and then later restrict attention to a subset of well-behaved terms (Section 3.1). The advantage of the mutual approach is that there are no intermediate "junk" terms, and that the more precise types often suggests necessary lemmas to prove. However this is mostly an ergonomic advantage, since the two approaches are equivalent (Section 3.4).

We also use the feature of higher inductive types [20] that has recently been added to Agda under the `--cubical` flag [30]. We define a different ordinal notation system for ordinals below ε_0 as a quotient inductive type [1], where we represent ordinals by finite hereditary multisets (Section 3.3). Path constructors are used to identify multiple representations of the same ordinal, so that we again recover uniqueness. Also this approach is equivalent to the other two approaches (Section 3.4).

Different representations are convenient for different purposes. For instance, the higher inductive type approach to define the ordinal notation system is convenient for defining

CPP '20, January 20–21, 2020, New Orleans, LA, USA

e.g. the commutative Hessenberg sum of ordinals (Section 4), while the mutual representation is convenient for proving transfinite induction (Section 5). Using the univalence principle [28], we can transport such constructions and properties between the different ordinal notation systems as needed.

Contributions We make the following contributions:

- We give two to our knowledge new ordinal notation systems in type theory, representing ordinals below ε_0. These can be used to verify *e.g.* termination of programs inside type-theory-based proof assistants such as Agda.
- We prove that our ordinal notation systems are equivalent, and also equivalent to a third, well-known ordinal notation system based on a predicate of being in Cantor normal form. This allows us to transport constructions and properties between them using the univalence principle.
- We prove that our ordinal notation systems allow the principle of transfinite induction. This, and the rest of the development, is completely computational and axiom-free, in particular we do not need to assume *e.g.* excluded middle or countable choice.
- In general, we show how recent features of Agda such as simultaneous definitions and higher inductive types can be used to obtain user-friendly constructions, and how to work around common pitfalls.

Agda Formalization Our full Agda development can be found at https://doi.org/10.5281/zenodo.3588624.

2 Cubical Agda

We start by giving a brief introduction to cubical Agda, an implementation of Cubical Type Theory [7] in the Agda proof assistant [23]. We refer to the Agda Wiki[1] and the Agda User Manual[2] for more resources on Agda, and to Vezzosi, Mörtberg and Abel [30] for the technical details of the cubical extension of type theory.

Agda has a hierarchy of *universes* called Sets. The Cubical Agda library[3] renames them to Types to avoid the confusion with the notion of set in Homotopy Type Theory [28]. The lowest universe is now called Type_0, and it lives in Type_1. More generally, there is a universe $\mathsf{Type}\ \ell : \mathsf{Type}\ (\mathsf{lsuc}\ \ell)$ for each $\ell : \mathsf{Level}$, where $\mathsf{lsuc} : \mathsf{Level} \to \mathsf{Level}$ is the successor function of universe levels.

We make use of Agda features such as *mixfix operators*, *implicit arguments* and *generalizable variables* to improve the readability of our Agda code. In turn, they work as follows: A mixfix operator may contain one or more name parts and one or more underscores _. When applied, its arguments go in place of the underscore. For instance, when using the Level maximum function $\_\sqcup\_ : \mathsf{Level} \to \mathsf{Level} \to \mathsf{Level}$, we can write $\ell \sqcup \ell'$ which is the same as $\_\sqcup\_\ \ell\ \ell'$. The _ symbol also has other usages: when an argument is not (explicitly) needed in a definition, or a term can be inferred by Agda's unifier, we can replace it by _. We can even omit _ using implicit arguments, which are declared using curly braces {}. For instance, if we define

$\mathsf{id} : \{\ell : \mathsf{Level}\}\ \{A : \mathsf{Type}\ \ell\} \to A \to A$
$\mathsf{id}\ a = a$

then id zero type-checks, because the type checker knows $\mathsf{zero} : \mathbb{N}$ and $\mathbb{N} : \mathsf{Type}_0$ and hence can infer that the implicit argument ℓ is lzero (the lowest Level), and that A is $\mathbb{N}$. To explicitly give an implicit argument, we just enclose it in curly braces. For example, we can also write $\mathsf{id}\ \{\_\}\ \{\mathbb{N}\}\ \mathsf{zero}$. We often want our types and functions to be universe polymorphic by adding Level arguments in the declaration as in the above example. We can further omit $\{\ell : \mathsf{Level}\}$ by using generalizable variables: throughout our Agda development, we declare

variable $\ell\ \ell'\ \ell''$: Level

and then bindings for them are inserted automatically in declarations where they are not bound explicitly. For instance, now the identity function can be declared as

$\mathsf{id} : \{A : \mathsf{Type}\ \ell\} \to A \to A$

where ℓ is implicitly universally quantified. We also use generalizable arguments for the different notions of ordinal terms considered in this paper.

Agda supports simultaneous definition of several mutually dependent data types such as in the schemes of *inductive-recursive* [13] and *inductive-inductive* [22] definitions. Both schemes permit the simultaneous definition of an inductive type A, together with a type family B over A; the difference between them lies in whether B is defined recursively over the inductive structure of A, or if B is itself inductively defined. The type A is allowed to refer to B and vice versa, so that one may for instance define A simultaneously with a predicate or relation B on A. In this paper, we will use this to define a type of ordinal notations simultaneously with their order relation (Section 3.2). The Agda syntax for mutual definitions is to place the type signature of all the mutually defined data types and/or functions before their definitions.

The cubical mode extends Agda with various features from Cubical Type Theory [7]. To use Agda's cubical mode, we have to place

{-# OPTIONS --cubical #-}

at the top of the file. First of all, cubical Agda has a primitive *interval* type I with two distinguished endpoints i0 and i1. Paths in a type A, representing equality between elements of A, are functions $\mathsf{I} \to A$; hence they can be introduced using λ-abstraction and eliminated using function application. There is a special primitive

[1] Agda Wiki: https://wiki.portal.chalmers.se/agda/pmwiki.php
[2] Agda User Manual: https://agda.readthedocs.io/
[3] Cubical Agda library: https://github.com/agda/cubical

```
PathP : (A : I → Set ℓ) → A i0 → A i1 → Set ℓ
```

which can be considered as the type of *dependent paths* whose endpoints are in different types. The type of non-dependent paths is defined by

```
_≡_ : {A : Set ℓ} → A → A → Set ℓ
_≡_ {A = A} = PathP (λ _ → A)
```

where {A = A} tells Agda to bind the implicit argument A declared in the type of _≡ _ to a variable also named A, which is used in the definition of _≡ _. In this paper, we will need the following path-related proofs from the cubical Agda library:

```
refl : x ≡ x
_⁻¹ : x ≡ y → y ≡ x
_•_ : x ≡ y → y ≡ z → x ≡ z
cong : (f : (a : A) → B a) (p : x ≡ y)
     → PathP (λ i → B (p i)) (f x) (f y)
cong₂ : (f : (a : A) → (b : B a) → C a b)
      → (p : x ≡ y)
      → {u : B x} {v : B y} (q : PathP (λ i → B (p i)) u v)
      → PathP (λ i → C (p i) (q i)) (f x u) (f y v)
transport : A ≡ X → A → X
subst : x ≡ y → B x → B y
```

A type is called a *proposition* if all its elements are identical, and is called a *set* if all its path spaces are propositions. In Agda, this is formulated as follows:

```
isProp isSet : Type ℓ → Type ℓ
isProp A = (x y : A) → x ≡ y
isSet A = {x y : A} → isProp (x ≡ y)
```

These univalent concepts play an important role in the development of mathematics in Homotopy Type Theory. Cubical Agda also supports a general schema of *higher inductive types* [9], a generalization of inductive types allowing constructors to produce paths. In this paper, we will construct an ordinal notation system as a higher inductive type (Section 3.3).

Another important concept from Homotopy Type Theory is the notion of type equivalence. We say that two types A and B are *equivalent*, and write $A \simeq B$, if there is a function $f : A \to B$ with an two-sided inverse $g : B \to A$, and if the proofs that f and g are inverses are coherent in a suitable sense. Importantly, every isomorphism (*i.e.* a function with a two-sided inverse, but without coherence conditions on the inverse proofs) gives rise to an equivalence, *i.e.* we have

```
isoToEquiv : Iso A B → A ≃ B
```

where we have written Iso A B for the type of isomorphisms between A and B. The *univalence principle* $(A \equiv B) \simeq (A \simeq B)$ is provable in cubical Agda. In particular, there is a function ua : $A \simeq B \to A \equiv B$ generating a path between two types from a proof that they are equivalent. We will use univalence to construct paths between equivalent systems of ordinal notations (Section 3.4) and then transport various constructions and proofs between them along these paths (Sections 4 and 5).

We will also use the following standard Agda data types:

- The empty type (with no constructors)

```
data ⊥ : Type₀ where
```

- Coproduct types (disjoint unions)

```
data _⊎_ (A : Type ℓ) (B : Type ℓ') : Type (ℓ ⊔ ℓ') where
  inj₁ : A → A ⊎ B
  inj₂ : B → A ⊎ B
```

- Σ-types (dependent pairs)

```
record Σ {A : Type ℓ} (B : A → Type ℓ') : Set (ℓ ⊔ ℓ') where
  constructor _,_
  field
    pr₁ : A
    pr₂ : B pr₁
```

- Cartesian products (non-dependent pairs)

```
_×_ : Type ℓ → Type ℓ' → Type (ℓ ⊔ ℓ')
A × B = Σ \(_ : A) → B
```

- The natural numbers, and the standard order relation on them

```
data ℕ : Type₀ where
  zero : ℕ
  suc : ℕ → ℕ

data _≤ᴺ_ : ℕ → ℕ → Type₀ where
  z≤n : {n : ℕ} → zero ≤ᴺ n
  s≤s : {n m : ℕ} → n ≤ᴺ m → suc n ≤ᴺ suc m
```

When the type of a variable x can be inferred, we will adopt the notational convention $\forall\ x \to P$ for $(x : \_) \to P$, and similarly $\forall\ \{x\} \to P$ for $\{x : \_\} \to P$.

When reasoning using chains of equations, we may write

```
begin
x ≡⟨ p ⟩
y ≡⟨ q ⟩
z □
```

for readability, where $p : x \equiv y$ and $q : y \equiv z$. This desugars to uses of transitivity $p \bullet q$, but has the advantage of keeping x, y and z explicit.

3 Notation Systems for Ordinals Below ε_0

The classical set-theoretic theory of ordinals defines an ordinal to be a set α which is transitive (*i.e.* $x \in \alpha \to x \subseteq \alpha$) and connected (*i.e.* $x \neq y \to x \in y \vee y \in x$ for any $x, y \in \alpha$). For program verification, the perhaps most important consequence of this definition is that $\in$ is a well ordering on ordinals — we hence often write $\alpha < \beta$ for $\alpha \in \beta$ — since

this implies that properties of ordinals can be proven by transfinite induction, which in turn implies that there can be no infinitely descending chains of ordinals

$$\alpha_0 > \alpha_1 > \alpha_2 > \dots$$

— in other words, any process that can be assigned a decreasing sequence of ordinals must terminate.

Obviously the empty set $\emptyset$ is an ordinal (commonly denoted 0), and if α is an ordinal, it is not hard to see that its *successor* $\alpha + 1 = \alpha \cup \{\alpha\}$ is also an ordinal. This way, we can construct all finite ordinals $1 = 0 + 1$, $2 = 1 + 1$, $3 = 2 + 1$, ..., and then take their limit $\omega = \{0, 1, 2, 3, \dots\}$. We can then continue constructing $\omega + 1$, $\omega + 2$, ..., eventually reaching $\omega + \omega = \omega \cdot 2$, then $\omega \cdot 3$, ... and thus eventually $\omega \cdot \omega = \omega^2$. Iterating this process, we can construct ω^ω, and then take the limit of the sequence

$$\omega^\omega, \omega^{\omega^\omega}, \omega^{\omega^{\omega^\omega}}, \dots$$

The resulting ordinal is denoted ε_0, and is the minimal ordinal α such that $\omega^\alpha = \alpha$. It is well known that every ordinal α can be written uniquely in so-called Cantor normal form

$$\alpha = \omega^{\beta_1} + \omega^{\beta_2} + \dots + \omega^{\beta_n} \qquad \text{with } \beta_1 \geq \beta_2 \geq \dots \geq \beta_n$$

for some natural number n and ordinals β_i (the special case $\alpha = 0$ is written as the empty sum with $n = 0$). Our primary interest in ε_0 is that if $\alpha < \varepsilon_0$, then every exponent β_i in the Cantor normal form of α satisfies $\beta_i < \alpha$. Hence if we in turn write $\beta_i = \omega^{\gamma_1} + \dots + \omega^{\gamma_m}$ in Cantor normal form, we discover a decreasing sequence

$$\alpha > \beta_i > \gamma_j > \dots$$

of ordinals, which hence must terminate in finitely many steps. As a result, we have a finitary *notation system* which we can hope to implement inside a theorem prover in order to represent the ideal concept of ordinals below ε_0 in it. In the rest of this section, we explore three different approaches for achieving this in Agda.

3.1 The Subset Approach SigmaOrd

Traditional approaches to ordinal notation systems such as Buchholz [5], Schütte [25] and Takeuti [27] usually start by generating ordinal terms inductively, and then single out a subset in order to provide a unique representation for ordinals. Along this direction, we construct a notation system of ordinals below ε_0 as a sigma type in an Agda module SigmaOrd.

The first step is to define ordinal terms, which are simply *binary trees*, albeit with highly suggestive constructor names:

```
data Tree : Type₀ where
  0 : Tree
  ω^_+_ : Tree → Tree → Tree
```

The idea is that 0 represents the ordinal 0, and ω^ a + b represents $\omega^\alpha + \beta$ if a and b represent α and β respectively. However $\omega^\alpha + \beta$ might not be in Cantor normal form, and might have multiple such representations, because no order constraint has been imposed in the exponents occurring in ω^ a + b. To remedy this flaw, we define an *ordering* on trees as follows (where a b c d : Tree):

```
data _<_ : Tree → Tree → Type₀ where
  <₁ : 0 < ω^ a + b
  <₂ : a < c → ω^ a + b < ω^ c + d
  <₃ : a ≡ c → b < d → ω^ a + b < ω^ c + d
```

The first constructor <₁ states that 0 is smaller than any other tree, and the constructors <₂ and <₃ say that non-0 trees are compared lexicographically. However, this is not a well-founded order on Tree! To recover well-foundedness, we must restrict to trees that are in Cantor normal form. Towards this, we define the non-strict order ≥ in terms of the strict order <:

```
_>_ _≥_ : Tree → Tree → Type₀
a > b = b < a
a ≥ b = a > b ⊎ a ≡ b
```

Then we can define the predicate of being in *Cantor normal form*: 0 is in normal form, and ω^ a + b is in normal form if also a and b are, and in addition a is greater than or equal to the first exponent in b, formally expressed using the following function:

```
fst : Tree → Tree
fst 0 = 0
fst (ω^ a + _) = a
```

We construct the predicate isCNF formally using the following indexed inductive definition:

```
data isCNF : Tree → Type₀ where
  0isCNF : isCNF 0
  ω^+isCNF : isCNF a → isCNF b → a ≥ fst b
             → isCNF (ω^ a + b)
```

For instance, if a b c d : Tree are in Cantor normal form and $a \geq b \geq c \geq d$, then isCNF (ω^ a + ω^ b + ω^ c + ω^ d + 0) is inhabited.

Finally, we can form the subset of trees in Cantor normal form by the following dependent pair type:

```
SigmaOrd : Type₀
SigmaOrd = Σ \(a : Tree) → isCNF a
```

We are justified in using the "subset" terminology, because we can prove that isCNF is proof-irrelevant, *i.e.*

```
isCNFIsPropValued : isProp (isCNF a)
```

the proof of which in turn relies on the following facts:

```
TreeIsSet : isSet Tree
<IsPropValued : isProp (a < b)
```

Therefore, equality on SigmaOrd is determined only by the Tree component, *i.e.* we can prove

```
SigmaOrd⁼ : {x y : SigmaOrd} → pr₁ x ≡ pr₁ y → x ≡ y
```

For the formal proofs, we refer to our Agda development. This approach gives a faithful representation of ordinals below ε_0, but it is sometimes inconvenient to work with, *e.g.* one has to explicitly prove that all operations preserve being in Cantor normal form. Agda's termination checker is often happier with curried functions, which further discourages use of SigmaOrd as a programming abstraction.

3.2 The Mutual Approach MutualOrd

Instead of considering an imprecise type of trees, including trees not in Cantor normal form that do not represent ordinals, we can use Agda's support for mutual definitions to directly generate trees in Cantor normal form only, by simultaneously defining ordinal terms and an ordering on them. The idea is to additionally require the term representing an ordinal a to be greater than or equal to the first exponent of the term representing an ordinal b when forming the term representing $\omega^a + b$. Hence we also need to define the operation which computes the first exponent of an ordinal term simultaneously. All in all, in a module MutualOrd we define

```
data MutualOrd : Type₀
data _<_ : MutualOrd → MutualOrd → Type₀
fst : MutualOrd → MutualOrd
```

simultaneously by

```
data MutualOrd where
  0 : MutualOrd
  ω^_+_[_] : (a b : MutualOrd) → a ≥ fst b → MutualOrd

data _<_ where
  <₁ : 0 < ω^ a + b [ r ]
  <₂ : a < c → ω^ a + b [ r ] < ω^ c + d [ s ]
  <₃ : a ≡ c → b < d → ω^ a + b [ r ] < ω^ c + d [ s ]

fst 0 = 0
fst (ω^ a + _ [ _ ]) = a
```

where we write $a \geq b = a > b \uplus a \equiv b$. Obviously this is very similar to the definitions in Section 3.1, but this time, *every* term of type MutualOrd satisfies the order constraint because of the third argument of the constructor ω^_+_[_]. This means that every term that we can form is already in Cantor normal form, and there is no need for a separate predicate.

Remark 3.1. Because of the coproduct hidden in the constructor argument $a \geq \mathrm{fst}\ b$, and the function fst occurring in it, MutualOrd is a nested [2] inductive-inductive-recursive [22] definition. However, by replacing the constructor with a coproduct argument by two constructors (one for each summand), and by defining the *graph* of fst inductively instead of the function itself recursively, it is possible to define an equivalent non-nested, non-inductive-recursive type. This justifies the soundness of our current definition.

Just like in the subset approach, we can prove that the order relation _<_ is proof-irrelevant, *i.e.* there is a proof of isProp $(a < b)$ for every a and b. However, because of the mutual nature of the definitions, the following facts has to be proved simultaneously:

```
MutualOrdIsSet : isSet MutualOrd
<IsPropValued : isProp (a < b)
MutualOrd⁼ : {r : a ≥ fst b} {s : c ≥ fst d} → a ≡ c → b ≡ d
             → ω^ a + b [ r ] ≡ ω^ c + d [ s ]
```

One advantage when working with a tighter type such as MutualOrd compared to the looser Tree is that the right lemma is often naturally suggested in the course of a construction: for example, when proving MutualOrdIsSet, the lemma MutualOrd⁼ falls more or less immediately out as required by one of the subgoals.

For later use in Section 4, we note that we can prove (constructively) that the ordering _<_ is trichotomous, *i.e.*

```
<-tri : (a b : MutualOrd) → a < b ⊎ a ≥ b
```

The proof is the same as a simpler proof for Tree from Section 3.1, except that we have to make essential use of MutualOrd⁼.

3.3 The Higher Inductive Approach HITOrd

In our third approach, an ordinal may have multiple representations, but we ensure that all of them are identical in the sense of Cubical Type Theory. We do this by defining a *higher inductive type*, which is given by freely generated terms and paths between them. Instead of representing an ordinal by a list of ordinal representations (the exponents in its Cantor normal form), where the order matters, we instead consider finite multisets of ordinal representations, where the order of elements does not matter. Such finite multisets can be defined in a first-order way as a higher inductive type, as in Licata [19]. Because the elements of the multiset again are ordinal representations, what we need is a higher inductive type of so-called finite hereditary multisets. We make the following definition in the module HITOrd:

```
data HITOrd : Type₀ where
  0 : HITOrd
  ω^_⊕_ : HITOrd → HITOrd → HITOrd
  swap : ∀ a b c → ω^ a ⊕ ω^ b ⊕ c ≡ ω^ b ⊕ ω^ a ⊕ c
  trunc : isSet HITOrd
```

This is a higher inductive type, since it is given by listing its generating term constructors 0 and ω^_⊕_, as well as its generating path constructors swap and trunc. Cubical Agda supports higher inductive types natively, and their soundness is guaranteed by the cubical sets model [9]. As hinted at by the name of the constructor ω^_⊕_, our intention for a term $\omega^a \oplus b$ is no longer to represent the non-commutative sum of ordinals $\omega^\alpha + \beta$ where α and β are represented by a and b respectively, but rather the commutative *Hessenberg*

sum $\omega^\alpha \oplus \beta$ (see Section 4.2). This is justified by the inclusion of the path constructor swap, which states that terms with permuted exponents are identical, as illustrated by the following example (using equational reasoning combinators from the end of Section 2):

```
example : (a b c : HITOrd)
  → ω^ a ⊕ ω^ b ⊕ ω^ c ⊕ 0 ≡ ω^ c ⊕ ω^ b ⊕ ω^ a ⊕ 0
example a b c = begin
  ω^ a ⊕ ω^ b ⊕ ω^ c ⊕ 0 ≡⟨ swap a b _ ⟩
  ω^ b ⊕ ω^ a ⊕ ω^ c ⊕ 0 ≡⟨ cong (ω^ b ⊕_) (swap a c _) ⟩
  ω^ b ⊕ ω^ c ⊕ ω^ a ⊕ 0 ≡⟨ swap b c _ ⟩
  ω^ c ⊕ ω^ b ⊕ ω^ a ⊕ 0 □
```

Adding just the swap constructor would result in a lack of higher-dimensional coherence (*e.g.* we would expect swap *a b c* • swap *b a c* to be the reflexivity path), and so we also include the trunc constructor which forces HITOrd to be a set. This means that we can prove the following recursion principle for HITOrd:

```
rec : {A : Type ℓ}
    → isSet A
    → A
    → (_★_ : A → A → A)
    → (∀ x y z → x ★ (y ★ z) ≡ y ★ (x ★ z))
    → HITOrd → A
```

This recursion principle states that there is a function from HITOrd to any other type A which is closed under the same "constructors" as HITOrd. In other words, to define a function HITOrd → A using the recursion principle, A needs to be a set, and one needs not only a point of A and an operator _★_ : $A \to A \to A$, but also a proof of a "swap" rule for _★_. This stops us from defining *e.g.* a function fst : HITOrd → HITOrd with fst (ω^ *a* ⊕ *b*) = *a* by *a* ★ *b* = *a*, since this would require

$$a = a \star (b \star c) \equiv (b \star a) \star c = b$$

for any a, b : HITOrd, which is clearly not true. In general, the recursion principle can be used to define *non-dependent* functions out of HITOrd that respect the additional path constructors (we will make use of this in Section 3.4). Similarly, to prove properties of HITOrd, we will make use of the following induction principle for propositions:

```
indProp : (P : HITOrd → Type ℓ)
        → (∀ {x} → isProp (P x))
        → P 0
        → (∀ {x y} → P x → P y → P (ω^ x ⊕ y))
        → ∀ x → P x
```

Since the motive $P\,x$ is a proposition for every x by assumption, we do not need to ask for any methods involving path constructors — there are no non-trivial paths in $P\,x$. Both the recursion principle and the induction principle for propositions are instances of the full induction principle, which can be proven by pattern matching in cubical Agda.

3.4 Equivalences Between the Three Approaches

We now wish to show that all three approaches are in fact equivalent, in the strong sense of Homotopy Type Theory. To show $A \simeq B$, it suffices to construct an isomorphism between A and B. Hence we construct isomorphisms between SigmaOrd and MutualOrd, and between MutualOrd and HITOrd. In a new module Equivalences, we import the previous modules:

```
open import SigmaOrd    as S
open import MutualOrd   as M
open import HITOrd      as H
```

Since many names are shared between the imported modules (*e.g.* both SigmaOrd and MutualOrd define _<_ and fst), we use the short module names S, M and H to qualify ambiguous names, *e.g.* we write _S.<_ and S.fst to refer to the concepts from SigmaOrd, and _M.<_ and M.fst for the ones from MutualOrd.

3.4.1 SigmaOrd is Equivalent to MutualOrd

We first construct a function T2M from SigmaOrd to MutualOrd. To help Agda's termination checker, we define T2M in curried form — in fact the first component a : Tree of the sigma type can even be kept implicit. Because MutualOrd is defined simultaneously with its ordering, when defining T2M we have to simultaneously prove that it is monotone:

```
T2M : {a : Tree} → isCNF a → MutualOrd
T2M[<] : {a b : Tree} {p : isCNF a} {q : isCNF b}
       → a S.< b → T2M p M.< T2M q
T2M[≥fst] : {a b : Tree} {p : isCNF a} (q : isCNF b)
          → a S.≥ S.fst b → T2M p M.≥ M.fst (T2M q)
```

We omit the easy proofs of T2M[<] and T2M[≥fst] here, but give the definition of T2M since it is computationally relevant:

```
T2M 0IsCNF = 0
T2M (ω^+IsCNF p q r) =
  ω^ (T2M p) + (T2M q) [ T2M[≥fst] q r ]
```

Remark 3.2. When implementing T2M[≥fst], we also need the curried equivalent T2M[≡] of SigmaOrd$^=$ specialised to the image of T2M, which can be defined using the path induction principle. Unfortunately, this detour trips up Agda's termination checker. We work around this by converting a given path to an inductively defined propositional equality using the following construction:

```
PropEqfromPath : {A : Set ℓ} {x y : A} → x ≡ y → x P.≡ y
PropEqfromPath {x = x} p = subst (x P.≡_) p P.refl
```

Here P is the builtin module defining propositional equality P.≡ as inductively generated by the constructor P.refl. With this in hand, we can pattern match directly on the produced propositional equality instead of using path induction

when implementing T2M[≡], which placates the termination checker:

```
T2M[≡] : {a b : Tree} {p : isCNF a} {q : isCNF b}
       → a ≡ b → T2M p ≡ T2M q
T2M[≡] a=b with PropEqfromPath a=b
T2M[≡] a=b | P.refl = cong T2M (isCNFIsPropValued _ _)
```

Hopefully the termination checker of cubical Agda will be fixed to accept a direct proof in future versions.

For the reverse direction, we convert MutualOrd to Tree, and then show that the resulting trees are in Cantor normal form:

```
M2T : MutualOrd → Tree
M2T 0 = 0
M2T (ω^ a + b [ _ ]) = ω^ (M2T a) + (M2T b)

isCNF[M2T] : (a : MutualOrd) → isCNF (M2T a)
isCNF[M2T] 0 = 0IsCNF
isCNF[M2T] (ω^ a + b [ r ]) =
  ω^+IsCNF (isCNF[M2T] a) (isCNF[M2T] b)
           (M2T[≥fst] b r)
```

We have omitted the easy proofs that M2T is monotone:

```
M2T[<] : {a b : MutualOrd}
       → a M.< b → M2T a S.< M2T b
M2T[≥fst] : {a : MutualOrd} (b : MutualOrd)
          → a M.≥ M.fst b → M2T a S.≥ S.fst (M2T b)
```

Putting all the pieces together, we can now define maps from MutualOrd to SigmaOrd and vice versa:

```
S2M : SigmaOrd → MutualOrd
S2M (a , p) = T2M p

M2S : MutualOrd → SigmaOrd
M2S a = (M2T a , isCNF[M2T] a)
```

The proofs that the two compositions of S2M and M2S are identities rely on the fact that the orderings are proof-irrelevant; more precisely, they use the lemmas SigmaOrd$^=$ and MutualOrd$^=$:

```
S2M2T=pr₁ : (a : SigmaOrd) → M2T (S2M a) ≡ pr₁ a
S2M2T=pr₁ (0 , 0IsCNF) = refl
S2M2T=pr₁ (ω^ a + b , ω^+IsCNF p q r) =
  cong₂ ω^_+_ (S2M2T=pr₁ (a , p)) (S2M2T=pr₁ (b , q))

S2M2S=id : (a : SigmaOrd) → M2S (S2M a) ≡ a
S2M2S=id a = SigmaOrd$^=$ (S2M2T=pr₁ a)

M2S2M=id : (a : MutualOrd) → S2M (M2S a) ≡ a
M2S2M=id 0 = refl
M2S2M=id (ω^ a + b [ _ ]) =
  MutualOrd$^=$ (M2S2M=id a) (M2S2M=id b)
```

Since every isomorphism can be extended to an equivalence (using isoToEquiv), and we have just constructed an isomorphism between SigmaOrd and MutualOrd, we have proven:

Theorem 3.3. *SigmaOrd and MutualOrd are equivalent,* i.e. *there is a proof S≃M : SigmaOrd ≃ MutualOrd.*

Using ua : $A \simeq B \to A \equiv B$, one direction of the univalence principle, we get a path from SigmaOrd to MutualOrd.

Corollary 3.4. *SigmaOrd and MutualOrd are identical,* i.e. *there is a proof S≡M : SigmaOrd ≡ MutualOrd.*

3.4.2 MutualOrd is Equivalent to HITOrd

A translation from MutualOrd to HITOrd is easy: we simply forget about the order witnesses.

```
M2H : MutualOrd → HITOrd
M2H 0 = 0
M2H (ω^ a + b [ _ ]) = ω^ (M2H a) ⊕ (M2H b)
```

The other direction is more interesting. We need a binary operation _★_ : MutualOrd → MutualOrd → MutualOrd satisfying the "swap" rule in order to use the recursion principle of HITOrd. For this purpose, we notice that both MutualOrd and HITOrd admit a list structure: 0 is the empty list; and in ω^ a + b [r] and ω^ a ⊕ b respectively, a is the head and b is the tail. All lists in MutualOrd are in descending order, while those in HITOrd are quotiented by permutations so that it is impossible to access the order of elements in HITOrd lists. Coming back to the binary operation _★_ with this list-structure intuition in mind, we see that _★_ needs to add its first argument (regarding it as an element) into its second (regarding it as a list) such that different orders of doing this result in the same list. One operation satisfying these requirements is list insertion. Again, we simultaneously need to prove that insert preserves the ordering, since MutualOrd is simultaneously defined with it.

```
insert : MutualOrd → MutualOrd → MutualOrd
≥fst-insert : {a b : MutualOrd} (c : MutualOrd)
            → b M.≥ M.fst c → a M.< b
            → b M.≥ M.fst (insert a c)
```

The insert function implements the standard algorithm for list insertion (slightly obfuscated by our choice of constructor names). Similarly the proof ≥fst-insert follows the same call structure to show that insert is order-preserving.

```
insert a 0 = ω^ a + 0 [ M.≥0 ]
insert a (ω^ b + c [ r ]) with <-tri a b
... | inj₁ a<b = ω^ b + insert a c [ ≥fst-insert c r a<b ]
... | inj₂ a≥b = ω^ a + ω^ b + c [ r ] [ a≥b ]

≥fst-insert {a} 0 _ a<b = inj₁ a<b
≥fst-insert {a} (ω^ c + d [ _ ]) b≥c a<b with <-tri a c
... | inj₁ a<c = b≥c
... | inj₂ a≥c = inj₁ a<b
```

Here M.≥0 is a proof that $a \geq 0$ for every a. Using that < is trichotomous, *i.e.* using <-tri to compare any two elements, we can prove that insert satisfies the swap rule:

```
insert-swap : (x y z : MutualOrd)
            → insert x (insert y z) ≡ insert y (insert x z)
```

Hence we can use insert and the recursion principle for HITOrd to define

```
H2M : HITOrd → MutualOrd
H2M = rec MutualOrdIsSet 0 insert insert-swap
```

and then show that M2H and H2M form an isomorphism (the step case is using equational reasoning combinators, as explained in Section 2):

```
M2H2M=id : (a : MutualOrd) → H2M (M2H a) ≡ a
M2H2M=id 0 = refl
M2H2M=id (ω^ a + b [ r ]) = begin
    H2M (M2H (ω^ a + b [ r ]))
  ≡⟨ refl ⟩
    H2M (ω^ (M2H a) ⊕ (M2H b))
  ≡⟨ refl ⟩
    insert (H2M (M2H a)) (H2M (M2H b))
  ≡⟨ cong₂ insert (M2H2M=id a) (M2H2M=id b) ⟩
    insert a b
  ≡⟨ insert-+ a b r ⟩
    ω^ a + b [ r ] □
```

We omit the easy proof of the lemma

```
insert-+ : (a b : MutualOrd) (r : a M.≥ M.fst b)
         → insert a b ≡ ω^ a + b [ r ]
```

used in the final step. For the other direction, we use the induction principle for propositions:

```
H2M2H=id : (a : HITOrd) → M2H (H2M a) ≡ a
H2M2H=id = indProp P trunc base step
  where
    P : HITOrd → Type₀
    P x = M2H (H2M x) ≡ x
    base : P 0
    base = refl
    step : ∀ {x y} → P x → P y → P (ω^ x ⊕ y)
    step {x} {y} p q = begin
        M2H (H2M (ω^ x ⊕ y))
      ≡⟨ insert-⊕ (H2M x) (H2M y) ⟩
        ω^ M2H (H2M x) ⊕ M2H (H2M y)
      ≡⟨ cong₂ ω^_⊕_ p q ⟩
        ω^ x ⊕ y □
```

This is using the following lemma:

```
insert-⊕ : (a b : MutualOrd)
         → M2H (insert a b) ≡ ω^ (M2H a) ⊕ (M2H b)
```

Putting everything together, we have proven:

Theorem 3.5. *MutualOrd and HITOrd are equivalent,* i.e. *there is a proof M≃H : MutualOrd ≃ HITOrd.*

Corollary 3.6. *MutualOrd and HITOrd are identical,* i.e. *there is a proof M≡H : MutualOrd ≡ HITOrd.*

4 Ordinal Arithmetic

In this section, we demonstrate the usability of our definitions by showing how well-known arithmetic operations can be defined on them. We have two quite different data structures representing ordinals below ε_0: hereditary descending lists MutualOrd and finite hereditary multisets HITOrd. It is more convenient and efficient to construct the ordinary arithmetic operations on MutualOrd, because comparing the "heads" suffices for the constructions rather than iterating through the whole ordinal terms. On the other hand, constructing the commutative arithmetic operations such as Hessenberg sums and products is easier and more natural on HITOrd, because orders do not play a role in the constructions. Hence we implement ordinary ordinal addition and multiplication on MutualOrd, and Hessenberg addition and multiplication on HITOrd. We prove some properties of the operations, and then transport the constructions and proofs between them using the path M≡H : MutualOrd ≡ HITOrd.

4.1 Ordinary Addition and Multiplication

Ordinal arithmetic extends addition and multiplication from the natural numbers to all ordinals, including transfinite ones. It is famously non-commutative: $1 + \omega = \omega$, but $\omega + 1 > \omega$. On MutualOrd, we have to define addition whilst simultaneously proving the property that it preserves the ordering.

```
_+_  : MutualOrd → MutualOrd → MutualOrd
≥fst+ : {a : MutualOrd} (b c : MutualOrd)
      → a ≥ fst b → a ≥ fst c → a ≥ fst (b + c)
```

The interesting case of this well-known algorithm, when both summands are non-zero, is guided by the fact that ordinals of the form ω^β are so-called additive principal ordinals, *i.e.* if $\gamma < \omega^\beta$ then $\gamma + \omega^\beta = \omega^\beta$ (after defining addition, this is not hard to prove for MutualOrd). In particular if $\alpha < \beta$, then $\omega^\alpha < \omega^\beta$ and hence $\omega^\alpha + \omega^\beta = \omega^\beta$. The proof that addition preserves the ordering again follows the same structure as addition itself.

```
0 + b = b
a + 0 = a
(ω^ a + c [ r ]) + (ω^ b + d [ s ]) with <-tri a b
... | inj₁ a<b = ω^ b + d [ s ]
... | inj₂ a≥b = ω^ a + (c + ω^ b + d [ s ]) [ ≥fst+ c _ r a≥b ]

≥fst+ 0 _ r s = s
≥fst+ (ω^ _ + _ [ _ ]) 0 r s = r
≥fst+ (ω^ b + _ [ _ ]) (ω^ c + _ [ _ ]) r s with <-tri b c
... | inj₁ b<c = s
... | inj₂ b≥c = r
```

The construction of an element of MutualOrd contains also a proof that it is in Cantor normal form. When implementing _+_ above, the construction (more precisely, the last case

when $a \geq b$) explicitly tells us what property of _+_ is required to show that the sum is in Cantor normal form, and we are led to prove this property simultaneously. In the traditional subset approach, one usually constructs addition on all ordinal terms, and then proves that it preserves Cantor normal form. However one has to figure out what property of addition is needed for the proof oneself. The above example of a "construction-guided" proof demonstrates one advantage of the mutual approach.

Moving from programs to proofs, consider the following type stating that a given binary operation is associative:

```
Assoc : (A : Type₀) → (A → A → A) → Type₀
Assoc A _★_ = ∀ a b c → a ★ (b ★ c) ≡ (a ★ b) ★ c
```

We can construct an easy but lengthy proof

```
+assoc : Assoc MutualOrd _+_
```

that _+_ on MutualOrd is associative — the lengthiness is due to the use of a case distinction on <-tri *a b* in the definition of _+_. Now, using the path M≡H : MutualOrd ≡ HITOrd, we can transport both the operation of addition and the proof that it is associative to an associative operation on HITOrd:

```
_+ᴴ_ : HITOrd → HITOrd → HITOrd
_+ᴴ_ = transport (λ i → M≡H i → M≡H i → M≡H i) _+_

+ᴴassoc : Assoc HITOrd _+ᴴ_
+ᴴassoc = transport (λ i → Assoc (M≡H i) (+Path i)) +assoc
```

where

```
+Path : PathP (λ i → M≡H i → M≡H i → M≡H i) _+_ _+ᴴ_
```

is a dependent path from _+_ to _+ᴴ_.

Similarly, we can implement the standard multiplication algorithm for ordinals in Cantor normal form

```
_·_ : MutualOrd → MutualOrd → MutualOrd
0 · b = 0
a · 0 = 0
a · (ω^ 0 + d [ r ]) = a + a · d
(ω^ a + c [ r ]) · (ω^ b + d [ s ]) =
  M.ω^⟨ a + b ⟩ + (ω^ a + c [ r ] · d)
```

where M.ω^⟨ *a* ⟩ = ω^ *a* + 0 [≥0]. Since every case is implemented in terms of previously defined functions, there is no need to prove any simultaneous lemma about preservation of the order this time. Again, we can transport this definition to get multiplication on HITOrd for free:

```
_·ᴴ_ : HITOrd → HITOrd → HITOrd
_·ᴴ_ = transport (λ i → M≡H i → M≡H i → M≡H i) _·_
```

Let us look at some examples. We define the MutualOrd representation of the ordinal 1 by M.1 = M.ω^⟨ 0 ⟩ and the one of ω by M.ω = M.ω^⟨ M.1 ⟩. The following examples illustrate that ordinal addition and multiplication are not commutative: for addition, we have $1 + \omega = \omega \neq \omega + 1$, where the equality is definitional, *i.e.*, it computes:

```
Ex[+NonComm] : M.1 + M.ω ≡ M.ω
             × M.ω + M.1 > M.ω
Ex[+NonComm] = (refl , <₃ refl <₁)
```

Similarly, for multiplication, we have $2 \cdot \omega = \omega \neq \omega + \omega = \omega \cdot 2$:

```
Ex[·NonComm] : (M.1 + M.1) · M.ω ≡ M.ω
             × M.ω               < M.ω + M.ω
             × M.ω + M.ω         ≡ M.ω · (M.1 + M.1)
Ex[·NonComm] = (refl , <₃ refl <₁ , refl)
```

For the examples of HITOrd, we define H.ω^⟨ *a* ⟩ = ω^ *a* ⊕ 0, H.1 = H.ω^⟨ 0 ⟩ and H.ω = H.ω^⟨ H.1 ⟩. The operations of addition and multiplication on HITOrd are obtained by transporting those on MutualOrd along M≡H. We get this path using (one direction of) the univalence axiom which is constructively provable in cubical Agda. Therefore, closed terms of HITOrd constructed using these operations can be evaluated into normal form, for instance

```
Ex[+ᴴComp] : H.1 +ᴴ H.ω ≡ ω^ (ω^ 0 ⊕ 0) ⊕ 0
Ex[+ᴴComp] = refl

Ex[·ᴴComp] : H.ω ·ᴴ (H.1 +ᴴ H.1)
           ≡ ω^ (ω^ 0 ⊕ 0) ⊕ ω^ (ω^ 0 ⊕ 0) ⊕ 0
Ex[·ᴴComp] = refl
```

Again, note that both equalities are definitional.

4.2 Hessenberg Addition and Multiplication

Hessenberg arithmetic [17] is a variant of ordinal arithmetic which is commutative and associative, but not continuous in its second argument. On HITOrd, Hessenberg addition is simply implemented as the concatenation operation on finite multisets. Here we define it by pattern matching on the first argument, which is equivalent to using the recursion principle. Note that we also have to produce clauses for swap and trunc, corresponding to proving that the defined function preserves the generating paths. For instance, for swap, we have to prove that our definition gives identical results for swapped exponents, *i.e.*, a path ω^ *a* ⊕ ω^ *b* ⊕ (*c* ⊕ *y*) ≡ ω^ *b* ⊕ ω^ *a* ⊕ (*c* ⊕ *y*), which is again an instance of swap:

```
_⊕_ : HITOrd → HITOrd → HITOrd
0              ⊕ y = y
(ω^ a ⊕ b)     ⊕ y = ω^ a ⊕ (b ⊕ y)
(swap a b c i) ⊕ y = swap a b (c ⊕ y) i
(trunc p q i j) ⊕ y = trunc (cong (_⊕ y) p) (cong (_⊕ y) q) i j
```

Our goal is now to justify the notation ⊕ in the constructor name for HITOrd by showing that _⊕_ is commutative. First we define the property of being commutative:

```
Comm : (A : Type₀) → (A → A → A) → Type₀
Comm A _★_ = ∀ a b → a ★ b ≡ b ★ a
```

Next we can use the induction principle for propositions to prove that indeed _⊕_ is commutative. The base case is given by a simple lemma

```
⊕unitr : (a : HITOrd) → a ⊕ 0 ≡ a
```

and the heavy work of the step case is done by the lemmas

```
⊕assoc : Assoc HITOrd _⊕_
ω^⊕=⊕ω^ : (a b : HITOrd) → (ω^ a ⊕ b) ≡ b ⊕ H.ω^⟨ a ⟩
```

which are also proved using the induction principle indProp. Using these lemmas, the proof is as follows:

```
⊕comm : Comm HITOrd _⊕_
⊕comm a = indProp P trunc base step
  where
    P : HITOrd → Type₀
    P b = a ⊕ b ≡ b ⊕ a
    base : P 0
    base = ⊕unitr a
    step : ∀ {x y} → P x → P y → P (ω^ x ⊕ y)
    step {x} {y} _ p = begin
        a ⊕ (ω^ x ⊕ y)
      ≡⟨ cong (a ⊕_) (ω^⊕=⊕ω^ x y) ⟩
        a ⊕ (y ⊕ H.ω^⟨ x ⟩)
      ≡⟨ ⊕assoc a y H.ω^⟨ x ⟩ ⟩
        (a ⊕ y) ⊕ H.ω^⟨ x ⟩
      ≡⟨ cong (_⊕ H.ω^⟨ x ⟩) p ⟩
        (y ⊕ a) ⊕ H.ω^⟨ x ⟩
      ≡⟨ (ω^⊕=⊕ω^ x (y ⊕ a)) ⁻¹ ⟩
        (ω^ x ⊕ y) ⊕ a □
```

By transporting along the reversed path

```
H≡M : HITOrd ≡ MutualOrd
H≡M i = M≡H (~ i)
```

we get a commutative operation on MutualOrd:

```
_⊕ᴹ_ : MutualOrd → MutualOrd → MutualOrd
_⊕ᴹ_ = transport (λ i → H≡M i → H≡M i → H≡M i) _⊕_

⊕ᴹcomm : Comm MutualOrd _⊕ᴹ_
⊕ᴹcomm = transport (λ i → Comm (H≡M i) (⊕Path i))
                  ⊕comm
```

where

```
⊕Path : PathP (λ i → H≡M i → H≡M i → H≡M i) _⊕_ _⊕ᴹ_
```

is a dependent path from _⊕_ to _⊕ᴹ_.

We also implement Hessenberg multiplication on HITOrd, which is essentially pairwise concatenation of elements in finite multisets. We first define $a \dot{+} b$ which concatenates every element of a with b. Again, we are asked to prove that this respects swapping exponents and set-truncation.

```
_∔_ : HITOrd → HITOrd → HITOrd
0              ∔ b = 0
(ω^ a ⊕ c)     ∔ b = ω^ (a ⊕ b) ⊕ (c ∔ b)
(swap x y z i) ∔ b = swap (x ⊕ b) (y ⊕ b) (z ∔ b) i
(trunc p q i j) ∔ b = trunc (cong (_∔ b) p) (cong (_∔ b) q) i j
```

Then we define Hessenberg multiplication $a \otimes b$ by using this operation to concatenate a to every exponent of b:

```
_⊗_ : HITOrd → HITOrd → HITOrd
a ⊗ 0          = 0
a ⊗ (ω^ b ⊕ c) = (a ∔ b) ⊕ (a ⊗ c)
a ⊗ (swap x y z i) = ⊕swap (a ∔ x) (a ∔ y) (a ⊗ z) i
a ⊗ (trunc p q i j) = trunc (cong (a ⊗_) p) (cong (a ⊗_) q) i j
```

where

```
⊕swap : ∀ a b c → a ⊕ b ⊕ c ≡ b ⊕ a ⊕ c
```

is easily proved using ⊕assoc and ⊕comm. Finally we can again transport to get Hessenberg multiplication on MutualOrd:

```
_⊗ᴹ_ : MutualOrd → MutualOrd → MutualOrd
_⊗ᴹ_ = transport (λ i → H≡M i → H≡M i → H≡M i) _⊗_
```

Let us look at some examples. Hessenberg addition on HITOrd can be viewed as a concatenation operation, as illustrated below:

```
Ex[⊕concat] :
     H.1  ⊕ H.ω^⟨ H.ω ⟩              ⊕ H.ω
  ≡ ω^ 0 ⊕ ω^ (ω^ (ω^ 0 ⊕ 0) ⊕ 0) ⊕ ω^ (ω^ 0 ⊕ 0) ⊕ 0
Ex[⊕concat] = refl
```

Again, because univalence is computational in cubical Agda, the transported Hessenberg operations on MutualOrd compute. For instance, we have the following definitional equalities — note that these equations are not true for ordinary addition and multiplication.

```
Ex[⊕ᴹComp] : M.1 ⊕ᴹ M.ω ≡ M.ω + M.1
Ex[⊕ᴹComp] = refl

Ex[⊗ᴹComp] : (M.1 + M.1) ⊗ᴹ M.ω ≡ M.ω + M.ω
Ex[⊗ᴹComp] = refl
```

5 Transfinite Induction

In this section, we prove transfinite induction for MutualOrd, and then transport it to transfinite induction for HITOrd. Already defining an ordering on HITOrd by hand is non-trivial, and usually requires several auxiliary concepts such as a subset relation for multisets and multiset operations such as union and subtraction [3, 11]. Now we can simply transport the ordering on MutualOrd to HITOrd. Similarly, it seems easier to prove transfinite induction for MutualOrd and then transport the proof to HITOrd if needed, rather than proving it directly.

5.1 The Transported Ordering on HITOrd

We firstly transport the ordering on MutualOrd to HITOrd as follows:

```
_<ᴴ_ : HITOrd → HITOrd → Type₀
_<ᴴ_ = transport (λ i → M≡H i → M≡H i → Type₀) _<_
```

We can further transport the properties of _<_ to _<^{H}_. For instance, let us define the property of decidability

```
Dec : (A : Type ℓ) → (A → A → Type ℓ') → Type (ℓ ⊔ ℓ')
Dec A _<_ = (x y : A) → x < y ⊎ ¬ x < y
```

We can easily prove

```
<-dec : Dec MutualOrd _<_
```

and then transport it to get

```
<H-dec : Dec HITOrd _<H_
<H-dec = transport (λ i → Dec (M≡H i) (<Path i)) <-dec
```

where

```
<Path : PathP (λ i → M≡H i → M≡H i → Type₀) _<_ _<H_
```

is a dependent path from _<_ to _<^{H}_.

Now we demonstrate that the transported property _<^{H}_ computes, like the transported constructions in Section 4. To simplify the examples, we turn <^{H}-dec into a boolean-valued function by

```
lt : HITOrd → HITOrd → Bool
lt a b = isLeft (<H-dec a b)
```

where isLeft assigns true : Bool to the left summand and false : Bool to the right. Here are some examples:

```
Ex[<H-decComp] :
    lt 0 0 ≡ false
  × lt Hω ((H1 ⊕ H1) ⊗ Hω) ≡ true
  × lt (Hω^⟨ Hω ⟩) (Hω^⟨ H1 +H Hω ⟩) ≡ false
  × lt (Hω^⟨ Hω ⟩) (Hω^⟨ H1 ⊕ Hω ⟩) ≡ true
Ex[<H-decComp] = (refl , refl , refl , refl)
```

Again, note that all equalities displayed are definitional.

5.2 Transfinite Induction

Transfinite induction for a type A with respect to a relation _<_ on A says that if for every x in A a property $P(x)$ is provable assuming that $P(y)$ holds for all $y < x$, then $P(x)$ holds for every x.

```
TI : (A : Type ℓ) → (A → A → Type ℓ') →
     ∀ ℓ'' → Type (ℓ ⊔ ℓ' ⊔ lsuc ℓ'')
TI A _<_ ℓ'' = (P : A → Type ℓ'')
             → (∀ x → (∀ y → y < x → P y) → P x)
             → ∀ x → P x
```

It is well-known that transfinite induction is logically equivalent to every element of A being accessible, in the following sense:

```
module Acc (A : Type ℓ) (_<_ : A → A → Type ℓ') where

  data isAccessible (x : A) : Type (ℓ ⊔ ℓ') where
    next : (∀ y → y < x → isAccessible y) → isAccessible x

  accInd : (P : A → Type ℓ'')
         → (∀ x → (∀ y → y < x → P y) → P x)
         → ∀ x → isAccessible x → P x
  accInd P step x (next δ) =
    step x (λ y r → accInd P step y (δ y r))

open Acc MutualOrd _<_
```

The proof of transfinite induction uses accInd. We now show that every element of MutualOrd is accessible:

```
WF : (x : MutualOrd) → isAccessible x
WF 0 = 0Acc
WF (ω^ a + b [ r ]) = ω+Acc a b r (WF a) (WF b)
```

The base case 0Acc : isAccessible 0 is trivial. We show the non-zero case

```
ω+Acc : (a b : MutualOrd) (r : a ≥ fst b)
  → isAccessible a → isAccessible b
  → isAccessible (ω^ a + b [ r ])
```

using the following two lemmas

```
fstAcc : ∀ {a b x} → isAccessible a → isAccessible b
  → x < a → (r : x ≥ fst b)
  → isAccessible (ω^ x + b [ r ])
sndAcc : ∀ {a b y} → isAccessible a → isAccessible b
  → y < b → (r : a ≥ fst y)
  → isAccessible (ω^ a + y [ r ])
```

which are simultaneously proved. The idea is that, to prove the accessibility of ω^ a + b [r], we have to show that z is accessible for any $z <$ ω^ a + b [r]. There are three cases: (1) If z is 0, then we are done. (2) If z is ω^ c + d [s] with $c < a$, then we use fstAcc. (3) If z is ω^ c + d [s] with $c \equiv a$ and $b < d$, then we use sndAcc.

Combining accInd and WF, we can now prove:

Theorem 5.1. *Transfinite induction holds for MutualOrd,* i.e. *there is a proof* `MTI : TI MutualOrd _<_ ℓ`.

Transporting along our path M≡H, we also have:

Corollary 5.2. *Transfinite induction holds for HITOrd,* i.e. *there is a proof* `HTI : TI HITOrd _<H_ ℓ`.

5.3 All Strictly Descending Sequences are Finite

Now we consider a simple application of transfinite induction: to prove that all strictly descending sequences of ordinals below ε_0 are finite. Formulating this faithfully in Agda is not easy when representing sequences as functions from the natural numbers, and one often ends up with the negative formulation "there is no strictly descending sequence" instead. One may replace finiteness by eventual zeroness, but this would contradict the strictly descending condition. As a stronger and *computational* formulation, we introduce the following notion:

```
pseudo-descending : (ℕ → MutualOrd) → Type₀
pseudo-descending f =
  ∀ i → f i > f (suc i) ⊎ (f i ≡ 0 × f (suc i) ≡ 0)
```

Note that it is not enough to require only $f\,i \equiv 0$ in the second summand, as that would allow f to "restart" at stage suc i.

This notion is obviously weaker than the notion of being strictly descending:

```
strictly-descending : (ℕ → MutualOrd) → Type₀
strictly-descending f = ∀ i → f i > f (suc i)
```

The following facts of pseudo-descendingness are trivial but play an important role in the proof.

```
zeroPoint : ∀ {f} → pseudo-descending f
  → ∀ {i} → f i ≡ 0 → ∀ j → j ≥ℕ i → f j ≡ 0
nonzeroPoint : ∀ {f} → pseudo-descending f
  → ∀ {i} → f i > 0 → f i > f (suc i)
```

where inequality $\_\leq^{\mathbb{N}}\_$ of natural numbers is inductively defined in the standard way. Moreover, we say that a sequence f is *eventually zero* if we can find an n such that $f(i)$ takes the value zero for every i after n:

```
eventually-zero : (ℕ → MutualOrd) → Type₀
eventually-zero f = Σ \(n : ℕ) → ∀ i → i ≥ℕ n → f i ≡ 0
```

One can easily prove the following fact of eventual-zeroness:

```
eventually-zero-cons :
  ∀ f → eventually-zero (f ∘ suc) → eventually-zero f
```

Now we can formulate our result positively as follows:

Theorem 5.3. *Every pseudo-descending sequence is eventually zero,* i.e. *there is a proof*

```
PD2EZ : ∀ f → pseudo-descending f → eventually-zero f.
```

Proof. We prove the statement using transfinite induction on $f\,0$, *i.e.* we use the following motive:

```
P : MutualOrd → Type₀
P a = ∀ f → pseudo-descending f → f 0 ≡ a
    → eventually-zero f
```

We have to prove the following induction step:

```
step : ∀ x → (∀ y → y < x → P y) → P x
step x h f df f0=x with ≥0 {f 0}
step x h f df f0=x | inj₁ f0>0 = goal
  where
    f1<x : f 1 < x
    f1<x = subst (f 1 <_) f0=x (nonzeroPoint df f0>0)
    ezfs : eventually-zero (f ∘ suc)
    ezfs = h (f 1) f1<x (f ∘ suc) (df ∘ suc) refl
    goal : eventually-zero f
    goal = eventually-zero-cons f ezfs
step x h f df f0=x | inj₂ f0=0 = goal
  where
    fi=0 : ∀ i → f i ≡ 0
    fi=0 i = zeroPoint df f0=0 i z≤n
    goal : eventually-zero f
    goal = 0 , λ i _ → fi=0 i
```

It consists of two cases: (1) If $f\,0 > 0$, then $f\,1 < x$ by the fact nonzeroPoint. Hence $f \circ \mathrm{suc}$ is eventually zero by the hypothesis h, and so is f by the fact eventually-zero-cons. (2) If $f\,0 \equiv 0$, then f is constantly zero by the fact zeroPoint. Hence we can take `PD2EZ f df = MTI P step (f 0) f df refl`. □

The algorithm encoded in the above proof checks the values of $f\,0$, $f\,1$, ... in turn, until it finds a zero point. By construction, it will thus find the least n such that $f\,i \equiv 0$ for all $i \geq^{\mathbb{N}} n$. The transfinite induction principle proves that this procedure is terminating, using the assumption of pseudo-descendingness.

Because strict descendingness implies the pseudo notion, the negative formulation is a simple corollary.

Corollary 5.4. *There is no strictly descending sequence,* i.e. *there is a proof* `NSDS : ∀ f → strictly-descending f → ⊥`.

6 Comparison with Related Work

In this section, we compare existing work with our development.

Trees as Ordinals The relationship between ordinals — especially ordinals below ε_0 — and various tree structures is of course well known, and more or less folklore. Dershowitz [10] gives an overview of different ordinal representations using finite trees, and Dershowitz and Reingold [12] construct binary trees using Lisp-like list structures. This is similar to our definition MutualOrd, but our systems provide *unique* representations of ordinals. Jervell [18] gives a clever total ordering on finite trees with ε_0 the supremum of all binary trees. It is not straightforward to encode and work with this ordering in a proof assistant.

Ordinals in Type Theory Surprisingly large ordinals can be constructed in basic Martin-Löf Type Theory with primitive type of (countable) ordinals, but no recursion principle for it. Coquand, Hancock and Setzer [8] show that already in this setting, one can reach $\phi_{\varepsilon_0}(0)$, where ϕ_α is the Veblen hierarchy. Hancock [16] uses a class of predicate transformers called lenses to give a clean proof of (half of) Hancock's conjecture: Martin-Löf Type Theory with n universes can reach $\phi_{\phi_{\ldots(0)}(0)}(0)$ with n nestings of $\phi_{\varepsilon_0}(0)$. In contrast, in our work we are not restricting ourselves to a spartan type theory, but try to take full advantage of all of Agda, with the goal of producing an easy-to-use representation. It is clear that we can draw much inspiration from this line of work when going beyond ε_0. See also Setzer [26] for a general overview of the ordinals that can be constructed in different variations of type theory.

Formalisations Several formalisations of ordinals and ordinal notation systems exist in the literature. Manolios and Vroon[21] represents ordinals below ε_0 in the ACL2 theorem prover, based on a variation of Cantor normal form with

$$\omega^{\beta_1} c_1 + \ldots + \omega^{\beta_n} c_n \quad \text{with } \beta_1 > \ldots > \beta_n \text{ and all } c_i \text{ finite}$$

This is similar to our SigmaOrd representation, except that there are no mechanical guarantees that given inputs actually are in Cantor normal form. They also provide algorithms for ordinal arithmetic and comparisons of ordinals, but their correctness proofs have to assume that the given inputs are in Cantor normal form. In contrast, it is not possible to construct ordinal terms not in Cantor normal form in our systems. Similarly, Castéran and Contejean [6] and Grimm [15] develop significant theories of ordinals below ε_0 in Coq, including arithmetic operations and transfinite induction. This is again similar to our SigmaOrd approach (a choice perhaps made because Coq to date does not support simultaneous definitions or higher inductive types, which are needed for the MutualOrd and HITOrd approaches respectively).

Finite Multisets In Isabelle/HOL, Blanchette, Fleury and Traytel [3] define an inductive datatype of hereditary multisets to represent ordinals below ε_0, similar to our HITOrd approach. The representation relies on the notion of multisets in Isabelle's standard library, which are defined as natural number-valued functions with a finite support. This can be constructively problematic, for instance when defining ordinal exponentiation. In contrast, our use of higher inductive types to define multisets means that our datatypes are reassuringly first-order. Because hereditary multisets are viewed as a subtype of nested multisets, the nested multiset ordering and its well-foundedness proof are "lifted" to the hereditary multisets using the sophisticated machinery in Isabelle. However, defining the nested multiset ordering [11] is non-trivial and proving its well-foundedness is challenging as admitted in [3]. In comparison, our ordinal notation system MutualOrd is convenient to work with for instance to prove its well-foundedness. By showing that it is equivalent to hereditary multisets HITOrd, we obtain also a well-foundedness proof for the latter.

7 Concluding Remarks

We have used modern features of cubical Agda such as simultaneous definitions and higher inductive types to faithfully represent ordinals below ε_0, and shown that our definitions are easy to work with by defining common operations on, and proofs about, our ordinal notation systems. Our development is fully constructive.

Of course, in the world of ordinals, ε_0 is tiny; already Martin-Löf Type Theory with W-types and only one universe has proof-theoretic strength well beyond ε_0 [26], and simultaneous inductive-recursive definitions are known to increase the proof-theoretic strength even further (a consequence of Hancock's Conjecture [16]). Similarly Lumsdaine and Shulman [20] show that adding recursive higher inductive types increases the power of type theory by considering in particular a higher inductive type encoding of a variation of Brouwer tree ordinals. To verify *e.g.* termination of programs exhausting the strength of such systems, one would have to define even stronger ordinal notation systems. We conjecture that powerful definitional principles such as simultaneous inductive-recursive definitions and higher inductive types — perhaps combined — can be used to faithfully represent also larger ordinals, and hence be useful for such program verification problems.

Acknowledgments

We thank Nicolai Kraus, Helmut Schwichtenberg, Ryota Akiyoshi, Nils Köpp, Masahiko Sato and Anders Mörtberg for many interesting and illuminating discussions, and the anonymous reviewers for their helpful suggestions and comments. This work was supported by funding from the Engineering and Physical Sciences Research Council [grant number EP/M016951/1], the Alexander von Humboldt Foundation, and the LMUexcellent initiative.

References

[1] Thorsten Altenkirch, Paolo Capriotti, Gabe Dijkstra, Nicolai Kraus, and Fredrik Nordvall Forsberg. 2018. Quotient inductive-inductive types. In *Foundations of Software Science and Computation Structures (Lecture Notes in Computer Science)*, Christel Baier and Ugo Dal Lago (Eds.), Vol. 10803. Springer, Heidelberg, Germany, 293–310.

[2] Richard Bird and Lambert Meertens. 1998. Nested datatypes. In *Mathematics of Program Construction (Lecture Notes in Computer Science)*, Johan Jeuring (Ed.), Vol. 1422. Springer, Heidelberg, Germany, 52–67.

[3] Jasmin Christian Blanchette, Mathias Fleury, and Dmitriy Traytel. 2017. Nested multisets, hereditary multisets, and syntactic ordinals in Isabelle/HOL. In *Formal Structures for Computation and Deduction (Leibniz International Proceedings in Informatics (LIPIcs))*, Dale Miller (Ed.), Vol. 84. Schloss Dagstuhl–Leibniz-Zentrum für Informatik, Dagstuhl, Germany, 11:1–11:18.

[4] Jasmin Christian Blanchette, Andrei Popescu, and Dmitriy Traytel. 2014. Cardinals in Isabelle/HOL. In *Interactive Theorem Proving (Lecture Notes in Computer Science)*, Gerwin Klein and Ruben Gamboa (Eds.), Vol. 8558. Springer, Heidelberg, Germany, 111–127.

[5] Wilfried Buchholz. 1991. Notation systems for infinitary derivations. *Archive for Mathematical Logic* 30 (1991), 227–296.

[6] Pierre Castéran and Evelyne Contejean. 2006. On ordinal notations. (2006). Available at http://coq.inria.fr/V8.2pl1/contribs/Cantor.html.

[7] Cyril Cohen, Thierry Coquand, Simon Huber, and Anders Mörtberg. 2018. Cubical Type Theory: a constructive interpretation of the Univalence Axiom. In *21st International Conference on Types for Proofs and Programs 2015 (Leibniz International Proceedings in Informatics (LIPIcs))*, Tarmo Uustalu (Ed.), Vol. 69. Schloss Dagstuhl–Leibniz-Zentrum für Informatik, Dagstuhl, Germany, 5:1–5:34.

[8] Thierry Coquand, Peter Hancock, and Anton Setzer. 1997. Ordinals in type theory. Invited talk for CSL '97.

[9] Thierry Coquand, Simon Huber, and Anders Mörtberg. 2018. On Higher Inductive Types in Cubical Type Theory. In *Logic in Computer Science*. ACM, New York, USA, 255–264.

[10] Nachum Dershowitz. 1993. Trees, ordinals and termination. In *Theory and Practice of Software Development (Lecture Notes in Computer Science)*, Marie-Claude Gaudel and Jean-Pierre Jouannaud (Eds.), Vol. 668. Springer, Heidelberg, Germany, 243–250.

[11] Nachum Dershowitz and Zohar Manna. 1979. Proving termination with multiset orderings. *Commun. ACM* 22, 8 (1979), 465–476.

[12] Nachum Dershowitz and Edward M. Reingold. 1992. Ordinal arithmetic with list structures. In *Logical Foundations of Computer Science (Lecture Notes in Computer Science)*, Anil Nerode and Michael Taitslin (Eds.),

Vol. 620. Springer, Heidelberg, Germany, 117–138.

[13] Peter Dybjer. 2000. A general formulation of simultaneous inductive-recursive definitions in type theory. *Journal of Symbolic Logic* 65, 2 (2000), 525–549.

[14] Robert W. Floyd. 1967. Assigning Meanings to Programs. In *Symposium on Applied Mathematics*, J.T. Schwartz (Ed.), Vol. 19. American Mathematical Society, Providence, USA, 19–32.

[15] José Grimm. 2013. *Implementation of three types of ordinals in Coq*. Technical Report RR-8407. INRIA. Available at https://hal.inria.fr/hal-00911710.

[16] Peter Hancock. 2000. *Ordinals and Interactive Programs*. Ph.D. Dissertation. University of Edinburgh.

[17] Gerhard Hessenberg. 1906. *Grundbegriffe der Mengenlehre*. Vol. 1. Vandenhoeck & Ruprecht, Göttingen, Germany.

[18] Herman Ruge Jervell. 2005. Finite Trees as Ordinals. In *New Computational Paradigms*, S. Barry Cooper, Benedikt Löwe, and Leen Torenvliet (Eds.). Springer, Heidelberg, Germany, 211–220.

[19] Dan Licata. 2014. What is Homotopy Type Theory? Invited talk at Coq Workshop 2014. Slides available at http://dlicata.web.wesleyan.edu/pubs/l14coq/l14coq.pdf.

[20] Peter Lefanu Lumsdaine and Michael Shulman. 2019. Semantics of higher inductive types. *Mathematical Proceedings of the Cambridge Philosophical Society* (2019), 1–50.

[21] Panagiotis Manolios and Daron Vroon. 2005. Ordinal arithmetic: algorithms and mechanization. *Journal of Automated Reasoning* 34, 4 (2005), 387–423.

[22] Fredrik Nordvall Forsberg. 2013. *Inductive-inductive definitions*. Ph.D. Dissertation. Swansea University.

[23] Ulf Norell. 2007. *Towards a practical programming language based on dependent type theory*. Ph.D. Dissertation. Chalmers University of Technology.

[24] Peter H. Schmitt. 2017. A mechanizable first-order theory of ordinals. In *Automated Reasoning with Analytic Tableaux and Related Methods (Lecture Notes in Computer Science)*, Renate Schmidt and Cláudia Nalon (Eds.), Vol. 10501. Springer, Heidelberg, Germany, 331–346.

[25] Kurt Schütte. 1977. *Proof Theory*. Springer, Heidelberg, Germany.

[26] Anton Setzer. 2004. Proof theory of Martin-Löf Type Theory – An overview. *Mathematiques et Sciences Humaines* 42 année, n^{o}165 (2004), 59–99.

[27] Gaisi Takeuti. 1987. *Proof Theory* (2 ed.). North-Holland Publishing Company, Amsterdam.

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

[29] Alan Turing. 1949. Checking a Large Routine. In *Report of a Conference on High Speed Automatic Calculating Machines*. University Mathematical Laboratory, Cambridge, UK, 67–69.

[30] Andrea Vezzosi, Anders Mörtberg, and Andreas Abel. 2019. Cubical Agda: a dependently typed programming language with univalence and higher inductive types. *Proceedings of the ACM on Programming Languages* 3, ICFP (2019), 87:1–87:29.