Encoding The Ordinals in Cedille

Alex Hubers, ahubers@uiowa.edu

December 11, 2020

1 Foreword

In this report, I will try to incremenetally introduce the concepts which form the foundation for an encoding of the ordinal numbers and their arithmetic in Cedille. We will start in Section 2 with a recitation of the underlying mathematics that build the ordinals, using [Jec06] and [Pin14] as reference.

I hope to find an optimum between completeness and conciseness; I want this section to be a good reference on the topic for those new to it, but I will also need to omit some proofs and concepts for sake of length. If you're already familiar with these mathematics, you will likely only need to skim section 2. I have done my best to consolidate the cited works into a minimally comprehensive set of definitions and theorems.

Once we've laid the mathematical groundwork, we'll move onto Church and Kleene's encoding of the ordinals in lambda calculi in Section 3, originally introduced in [CK37] for the untyped lambda calculus. The encoding will serve principally as an introduction to the intuition by which we add limit ordinals into a lambda encoding. Thereafter we will use this intuition to define the ordinals as a type in System F.

Finally, in Section 4, we will demonstrate such an encoding in Cedille. We'll start with a Cedille translation of the development by Escardo in Agda [Esc11]. We'll then demonstrate a reimplementation using Cedille's impredicativity to avoid some of the roadblocks that Escardo encounters in Agda.

The source for section 4 can be found at
`https://www.github.com/ahubers/cedille-ordinals/`.

This report's intended audience should be an equivalence class of students from our Lambda Calculus course. That is to say, the reader should have a reasonable understanding of:

1. Basic Set Theory and Notation
2. untyped λ-calculus
3. the simply typed λ-calculus

Since it was covered in class, I'm also going to assume the reader has a minimal familiarity of System F – namely that it permits $\forall$-quantification over type variables (and has inherent impredicativity). Explanations are omitted because these topics were covered in class.

I will not presuppose any knowledge of basic order theory, higher ordered lambda calculi (e.g, dependent typing), or Cedille. Consequently, with respect to the scope of what I

can reasonably cover, I hope the definitions and concepts I introduce will be necessary and sufficient for comprehension of the report, assuming the reader is in our metaphorical equivalence class.

2 A Set Theoretic Foundation of the Ordinals

2.1 Partial and Well-Ordered sets

In this section I am going to build the ordinals from the ground up, starting with *strict partial orders*.

Definition 2.1 (Strict Partial-Ordering)**.** A binary relation $<$ on a set P is a *strict partial-ordering* of P if the following hold:

1. *(irreflexivity)* $p \not< p$ for all $p \in P$ (where $a \not< b$ signifies $\neg(a < b)$).
2. *(transitivity)* If $p < q$ and $q < r$, then $p < r$ for all $p, q, r \in P$.

Moreover, the pair $(P, <)$ is called a *partially ordered set*, or *poset*.

I'd like to note that some definitions of *strict partial orders* come included with an *asymmetry* property. I hope to reconcile this discrepancy by showing that asymmetry is in fact implied by our first two properties.

Lemma 2.1. Strict partial orders are asymmetric: if $a < b$ then $b \not< a$, for all a, b in some poset $(P, <)$.

Proof. Suppose $a < b$. The two are distinct because were a to equal b, then $a = b \Rightarrow a < a$, contradicting irreflexivity. Now suppose $b < a$. Then $a < b$ and $b < a$, by transitivity, implies $a < a$, again contradicting irreflexivity. Thus $\neg(b < a)$, which is how we defined $b \not< a$. □

Note that this definition starts with *strict partial-orderings* and consequently differs from the regular definition you may be accustomed to, defined below:

Definition 2.2 (Non-Strict Partial-Ordering)**.** Another common way to define a *partially ordered set* is a pair $(P, \leq)$ such that:

1. *(reflexivity)* $p \leq p$ for all $p \in P$.
2. *(transitivity)* If $p \leq q$ and $q \leq r$, then $p \leq r$.
3. *(anti-symmetry)* if $p \leq q$ and $q \leq p$, then $p = q$.

We can connect the two definitions by building the relation $\leq$ from our definition of the *partial-ordering* $<$, such that for all p, $q \in P$, $p \leq q$ iff $p = q$ or $p < q$. The distinction here is that our strict partial-ordering $<$ has *irreflexivity* and *asymmetry*, while $\leq$ is a *non-strict partial-ordering* with *reflexivity* and *anti-symmetry* instead.

Before introducing well-orders, we need to extend our *partial-orders* with a stronger property:

Definition 2.3 (Total-Ordering)**.** A given *strict partial-ordering* over P is a *total-ordering* (or *total order*) if the additional property holds:

1. *(Trichotomy)* for all $p, q \in P$, either $p < q$ or $q < p$ or $p = q$.

You can think of the trichotomy law as stating that in a *total order*, all elements should be comparable. As a fun, illustrative example, let's take the set $\{1, 2, 3, A, B\}$ and assume that $1, 2, 3$ are ordered as expected, $A < B$, and neither $A < A$ or $B < B$. This is actually a partial order! it's transitive and irreflexive, by definition. However we have no idea how to compare the numeral elements with A or B.

For the rest of this report, when I refer to an arbitrary partial-ordering $<$, it should be assumed that we are referring to a *strict partial-ordering* as earlier defined, which should bear the sufficient conditions for the rest of our definitions. If we wish to refer to a *reflexive partial-ordering*, I'll use the meta-symbol $\leq$ and try to make this distinction clear. In some cases, we will use $\leq$ simply for notational convenience.

As a note, at this point there is a plethora of properties one could start enumerating simply about posets. For the sake of being concise, I am going to only define what we will need for this report; for a fuller treatment, see definition 2.2 of [Jec06].

For the definitions that follow, assume that $(P, <)$ be is a partially ordered set, and let X be a nonempty subset of P.

Definition 2.4 (Least Element)**.** The *least element* of X, if it exists, is an element $a \in X$ such that $\forall x \in X$, $a \leq x$.

Definition 2.5 (upper bound)**.** An *upper bound* of X is an element $a \in P$ such that $x \leq a$ for all $x \in X$.

Definition 2.6 (Supremum)**.** The *supremum* of X, denoted often as $sup(X)$, is the least element $a \in P$ such that a is an upper bound of X. For this reason, the supremum is often also called the *least upper bound* of X.

With this in mind, we can define a *well-ordering* as:

Definition 2.7 (Well-Ordering)**.** For total-ordered set $(P, <)$, $<$ is a *well-ordering* (or *well-order*) if every nonempty subset of P has a *least element*. In such a case, we would call $(P, <)$ a *well-ordered set*.

Given two well ordered sets, we often want to ask if they are the same with respect to their ordering. Such a concept is called an *isomorphism*, and is central to most areas of mathematics.

For example, the set $\{1, 2, 3\}$, with respect to the partial ordering $<$, is isomorphic to $\{4, 5, 6\}$ – meaning that as far as the $<$ symbol cares, they're the same.

We will formalize this intuition below.

Definition 2.8 (bijection)**.** For sets A and B, a function $f : A \to B$ is called *bijective* if it satisfies the following:

1. (injectivity) for all $a, a' \in A$, if $f(a) = f(a')$ then $a = a'$.
2. (surjectivity) for all $b \in B$, there exists $a \in A$ such that $f(a) = b$.

A consequence of bijectivity is that f is invertible, meaning there exists $f^{-1} : B \to A$ such that $\forall a \in A$ and $\forall b \in B$, $f^{-1}(f(a)) = a$ and $f(f^{-1}(b)) = b$. A proof of this fact is omitted, but should be easy to deduce on one's own.

Definition 2.9 (monotonicity)**.** For two posets $(P, <)$ and $(Q, \sqsubset)$, a function $f : P \to Q$ is *monotonic* if, for all $x \in P$ and $y \in Q$, $x < y$ implies $f(x) \sqsubset f(y)$.

Definition 2.10 (isomorphism)**.** A bijective function $f : P \to Q$ is an *isomorphism* if f and f^{-1} are monotonic. Equivalently, you can also say that f is an isomorphism if $x < y \Leftrightarrow f(x) \sqsubset f(y)$ for all $x, y \in P$.

If such an f exists, we say that the posets $(P, <)$ and $(Q, \sqsubset)$ are *order isomorphic*. You can think of this as saying, "if I take two elements in P and move them over to Q using f (or vice-versa), they will be ordered the same way".

Although these definitions are defined on posets, they extend as one would expect to well-ordered sets.

2.2 The Ordinal Numbers

For the following section, we're going to define ordinal numbers as sets observing some desired properties. We'll then informally construct a canonical representation of such sets by extending Von Neumann's construction of the Naturals.

Definition 2.11 (Set Transitivity)**.** A set T is *transitive* if every element of T is a subset of T.

Definition 2.12 ($\in$-ordering)**.** A set P is said to be $\in$-ordered if it is partially ordered by the relation $\in$. This is to say, for $p, q, r \in P$:

1. (irreflexivity) $p \notin p$.
2. (Transitivity) $p \in q$ and $q \in r \Rightarrow p \in r$.

Definition 2.13 (Ordinal Number)**.** A set is an *ordinal number* if it is *transitive* and *well-ordered* by $\in$.

Note here that when we say an ordinal number is well-ordered by $\in$, we mean it satisifies definition 2.12 and, additionally, such an ordering is a well-order: all elements are comparable, and every nonempty subset has a least element.

With a definition of the ordinals and isomorphisms in hand, it's important we recognize the following theorem:

Theorem 2.2. Every well-ordered set is isomorphic to a unique ordinal number. We denote the ordinal to which such a set is isomorphic as the *order type* of that set.

I've omitted a proof, as it warrants a sleuth of definitions and additional lemmas that aren't otherwise particularly useful to our goal (this being a lambda encoding of the

ordinals). However, it's nevertheless very important. It means that if we have some well-ordered set, we may canonically represent it as an ordinal. This permits us to speak of well-orders as ordinals, and to speak about all ordinals through some canonical form.

2.2.1 Constructing the Ordinals

Von Neumann originally proposed a construction of the natural numbers as follows:

$$\begin{aligned} 0 &= \emptyset \\ n+1 &= n \cup \{n\} \end{aligned}$$

For example:

$$\begin{aligned} 0 &= \emptyset \\ 1 &= 0 \cup \{0\} = \{\emptyset\} \\ 2 &= 1 \cup \{1\} = \{\emptyset, \{\emptyset\}\} \end{aligned}$$

Such a definition gives us an aforementioned canonical form for the natural numbers.

Von Neumann extended his definition of the naturals to ordinals accordingly: each ordinal is the well-ordered set of all smaller ordinals. Symbolically,

$$\begin{aligned} 0 &= \emptyset \\ \alpha+1 &= \alpha \cup \{\alpha\} \end{aligned}$$

where now we may extend this definition for zero and the successor to limit ordinals – a limit ordinal being a ordinal which isn't a successor. The first such (nonzero) limit ordinal is denoted ω.

$$\begin{aligned} 0 &= \emptyset \\ 1 &= 0 \cup \{0\} = \{\emptyset\} \\ 2 &= 1 \cup \{1\} = \{\emptyset, \{\emptyset\}\} \\ &... \\ \omega &= sup(\{0, 1, 2, ...\}) \\ \omega+1 &= \omega \cup \{\omega\} = \{0, 1, 2, ..., \omega\} \end{aligned}$$

Here is where things start to become stranger than the natural numbers. If we have ω, what's the smallest ordinal bigger than it? Because ω is, after all, an ordinal, we can

simply take its successor: $\omega + 1$. Continuing this chain, we get the increasing sequence:

$$\omega < \omega + 1 < \omega + 2 < ... < \omega + \omega$$

At this point we've reached our second smallest nonzero limit ordinal: $\omega + \omega$, which we can denote as $\omega \cdot 2$. It consequently follows that we have another chain:

$$\omega < \omega + 1 < ... < \omega \cdot 2 < \omega \cdot 2 + 1 < ... < \omega \cdot 3 < ... < \omega \cdot \omega$$

where $\omega \cdot \omega$ is, as paralled by the cardinals, similarly denoted ω^2. Taking things further:

$$\omega^2 < \omega^3 < ... < \omega^\omega$$

This forms the basic intuition of how to get exceedingly bigger and bigger ordinals, where we'll ultimately get $\epsilon_0 = \omega^{\omega^{\omega^{...}}}$.

2.3 Successor and Limit Ordinals

While the above serves as a helpful illustration, we can more formally define successor and limit ordinals.

Definition 2.14 (Successor of α)**.** for given ordinal α, the successor of α, denoted $\alpha + 1$, is given by $\alpha \cup \{\alpha\}$.

Definition 2.15 (Limit Ordinals)**.** if there exists some ordinal β such that $\alpha = \beta + 1$, then α is a *successor ordinal*. If α is *not* a successor ordinal, then it is called a *limit ordinal*.

We're already very familiar with the first limit ordinal – it's 0, as one would expect.

Definition 2.16 (ω)**.** We denote the least nonzero limit ordinal as ω. The ordinals less than ω are called *finite ordinals*.

As a consequence of this definition, we can separate the ordinals in our minds as either being *finite* or *transfinite*. You can view the finite ordinals simply as natural numbers – they are either zero, or the successor of some natural number. Consequently, we may view the ordinals as being exclusively one of the following:

1. Zero
2. The successor of an ordinal
3. A nonzero limit ordinal

This intuition forms the basis of transfinite induction and recursion.

2.4 Transfinite Induction

I'd like to first draw a parallel to induction of the natural numbers. My reasoning for doing so is two-fold: firstly, that you may think of transfinite induction as an *extension* of the principle of induction to the ordinals. Additionally, the definition that follows of transfinite induction may seem odd or unfamiliar – namely that we show some class C is the class of all ordinals. This may feel odd if you are used to the principle of induction looking like this:

Definition 2.17 (Principle of Induction)**.** Suppose for some predicate P, that $P(0)$ is true and $P(n) \Rightarrow P(n+1)$ for all $n \in \mathbb{N}$. Then P is true for all n.

It can be shown that this statement is in fact logically equivalent to the more general statement, as follows:

Definition 2.18 (Principle of Induction)**.** Suppose that $A \subseteq \mathbb{N}$, that $0 \in A$, and that $n \in A$ implies $n+1 \in A$ for all $n \in \mathbb{N}$. Then $A = \mathbb{N}$.

The proof that these two are logically equivalent is omitted, but can be found in Appendix A of [JJ98].

Likewise, for the definition of transfinite induction that follows, I will phrase it in its general form, and omit a proof that this definition can be extended to propositions P over the ordinals; however, it can be assumed that we may use induction like we do on the naturals for the same reason.

With that detour taken:

Definition 2.19 (Principle of Transfinite Induction)**.** Let C be a class of ordinals. Assume that:

1. $0 \in C$.
2. if $\alpha \in C$, then $\alpha + 1 \in C$.
3. if α is a nonzero limit ordinal and $\beta \in C$ for all $\beta < \alpha$, then $\alpha \in C$.

Then C is the class of all ordinals.

Like with the naturals, we can reformulate over propositions as such:

Definition 2.20 (Principle of Transfinite Induction)**.** Let P be a proposition on ordinal numbers, and assume that:

1. $P(0)$.
2. if $P(\alpha)$, then $P(\alpha + 1)$.
3. if α is a nonzero limit ordinal and $P(\beta)$ for all $\beta < \alpha$, then $P(\alpha)$.

Then $P(\alpha)$ for any ordinal α.

You can liken this principle to finite induction by noting that they're really the same, except we have one additional case, which we'll denote as *the limit case*. Consequently, to prove P using transfinite induction, you must prove the first two cases as you would normally, and then additionally prove the limit case.

2.5 Transfinite Recursion

In induction, the objective is to prove a proposition P over all ordinals. For recursion, we wish to assign a value to a function $F(\alpha)$ for all α. Having this capability will allow us to define operations on the ordinals fairly cleanly.

Theorem 2.3 (Transfinite Recursion)**.** For notational convenience, denote the class of all ordinals as OR[1]. let $A \subseteq V$ be any class of sets, let $G : A \to A$ be a function, and let $a \in A$. Then there exists a unique function $F : OR \to A$ such that:

1. $F(0) = a$.
2. $F(\alpha + 1) = G(F(\alpha))$.
3. $F(\alpha) = \bigcup\{F(\beta) \mid \beta < \alpha\}$ if α is a limit ordinal.

2.6 Ordinal arithmetic

We can define operators on our ordinals using the transfinite recursion theorem. I will define addition by this method, then rephrase it and the other operators more concisely.

Definition 2.21 (Ordinal Addition)**.** For any arbitrary ordinal α, let $G(\alpha) = \alpha + 1$ be the ordinal successor function, and define a function $\sigma_\alpha : OR \to OR$ as follows:

1. (base case): $\sigma_\alpha(0) = \alpha$
2. (step case): $\sigma_\alpha(\beta + 1) = G(\sigma_\alpha(\beta)) = \sigma_\alpha(\beta) + 1$
3. (limit case): $\sigma_\alpha(\beta) = sup(\{\sigma_\alpha(\gamma) \mid \gamma < \beta\})$ if $\beta > 0$ is a limit ordinal.

Notationally, we can rephrase addition more clearly (but still relying on the underpinnings of transfinite recursion), as such:

Definition 2.22 (Ordinal Addition, again)**.** For all ordinals α:

1. $\alpha + 0 = \alpha$
2. $\alpha + (\beta + 1) = (\alpha + \beta) + 1$
3. $\alpha + \beta = sup\{\alpha + \gamma \mid \gamma < \beta\}$ for all limit ordinals $\beta > 0$.

Via transfinite induction, we can show these two definitions to be equivalent.

Theorem 2.4. For any ordinals α and β, $\sigma_\alpha(\beta) = \alpha + \beta$.

Proof. By transfinite induction on β.

Base case: if $\beta = 0$, then $\sigma_\alpha(0) = \alpha = \alpha + 0$.

[1] To limit the scope of this report, I've omitted (or willfully avoided) any definition or distinction between classes and sets. However it can be easily shown that there is no such set of all ordinals, and the class of all ordinals is actually a proper class. For the interested reader, I would recommend [Pin14] for a deeper dive into axiomatic set theory. For our purposes, just assume we may quantify over the class of all Ordinals, and denote that class as OR.

Step case: suppose β is a successor ordinal. Then there exists γ such that $\beta = \gamma + 1$. Consequently, $\sigma_\alpha(\gamma + 1) = \sigma_\alpha(\gamma) + 1 = (\alpha + \gamma) + 1 = \alpha + (\gamma + 1) = \alpha + \beta$, where we invoke our induction hypothesis to show that $\sigma_\alpha(\gamma) = \alpha + \gamma$.

Limit case: Suppose the theorem holds for all $\gamma < \beta$ and β is a limit ordinal. then $\sigma_\alpha(\beta) = sup\{\sigma_\alpha(\gamma) \mid \gamma < \beta\}$, which by our induction hypothesis equals $sup\{\alpha + \gamma \mid \gamma < \beta\} = \alpha + \beta$. □

We can define multiplication and exponentiation in a similarly abridged fashion as we did for addition.

Definition 2.23 (Ordinal Multiplication)**.** For all ordinals α:

1. $\alpha \cdot 0 = 0$
2. $\alpha + (\beta + 1) = \alpha \cdot \beta + \alpha$
3. $\alpha + \beta = sup\{\alpha \cdot \gamma \mid \gamma < \beta\}$ for all limit ordinals $\beta > 0$.

Definition 2.24 (Ordinal Exponentiation)**.** For all ordinals α:

1. $\alpha^0 = 1$.
2. $\alpha^{\beta+1} = \alpha^\beta \cdot \alpha$.
3. $\alpha^\beta = sup\{\alpha^\gamma \mid \gamma < \beta\}$ for all limit ordinals $\beta > 0$.

Remark. It is common as well to show that these formulations of addition and multiplication are isomorphic, respectively, to the *ordinal sum* and *ordinal product* of two disjoint well-ordered sets. I've omitted these definitions and theorems, as our encoding of the ordinals will rely on transfinite recursion and induction and not strictly relate to any set theoretic foundation. For the interested reader, though, see [Pin14], chapters 9 and 10, or Lemma 2.24 of [Jec06].

2.6.1 Associativity and Right-Distributivity

The ordinals observe some properties we're familiar with from the cardinals, however they disobey some others. Notably, addition and multiplication are associative, and multiplication distributes *on the right* through addition.

However, addition and multiplication *are not* commutative. For example, consider $\omega + 1 = \omega \cup \{\omega\}$, whereas $1 + \omega = 1 + sup\{1 + \gamma \mid \gamma < \omega\} = \omega$. Consequently $1 + \omega \neq \omega + 1$.

Lemma 2.5 (Associativity of addition)**.** $(\alpha + \beta) + \gamma = \alpha + (\beta + \gamma)$ for all ordinals α, β, γ.

Proof. By transfinite induction on γ.

base case: if $\gamma = 0$, the result is trivial.

step case: Consider $\gamma + 1$ and let m denote $(\alpha + \beta)$. Then $(\alpha + \beta) + (\gamma + 1) = m + (\gamma + 1)$. By definition, this reassociates to $(m + \gamma) + 1 = ((\alpha + \beta) + \gamma) + 1$. We may now invoke our induction hypothesis to say $(\alpha + (\beta + \gamma)) + 1$. For clarity, let $n = \beta + \gamma$. Then

$(\alpha + (\beta + \gamma)) + 1 = (\alpha + n) + 1 = \alpha + (n + 1) = \alpha + ((\beta + \gamma) + 1) = \alpha + (\beta + (\gamma + 1))$, as desired.

Limit case: Suppose γ is a limit ordinal and the theorem holds for all $\zeta < \gamma$. Then $(\alpha + \beta) + \gamma = sup\{(\alpha + \beta) + \zeta \mid \zeta < \gamma\} = sup\{\alpha + (\beta + \zeta) \mid \zeta < \gamma\} = \alpha + (\beta + \gamma)$ by induction hypothesis. □

The next two properties can be proven by induction on γ.

Lemma 2.6 (Associativity of multiplication)**.** $(\alpha \cdot \beta) \cdot \gamma = \alpha \cdot (\beta \cdot \gamma)$ for all ordinals α, β, γ.

Lemma 2.7 (Right distributivity)**.** $(\alpha + \beta) \cdot \gamma = \alpha \cdot \gamma + \beta \cdot \gamma$ for all ordinals α, β, γ.

2.7 Cantor Normal Form

Theorem 2.8 (Cantor's Normal Form Theorem)**.** Every ordinal $\alpha > 0$ can be represented uniquely in the form

$$\alpha = \omega^{\beta_1} \cdot k_1 + ... + \omega^{\beta_n} \cdot k_n$$

where $n \geq 1$, $\alpha \geq \beta_1 \geq ... \geq \beta_n$, and $k_1, ..., k_n$ are nonzero natural numbers.

This theorem, in English, just says that we can represent every ordinal using only the operations we've thus defined.

You may also think of the theorem as saying that all ordinals are representable with base ω – much in the same way we represent natural numbers in decimal form, e.g:

$$n = k_m \cdot 10^m + k_{m-1} \cdot 10^{m-1} + ... + k_0 \cdot 10^0$$

for any n, m, and $k_1, ..., k_m \in \mathbb{N}$.

We unfortunately do not make use of Cantor's Normal Form in our encoding, however it's a useful property to know, and it might have some application were the interested reader to explore it.

2.8 To ϵ_0, and beyond!

In section 2.2.1 we illustrated an ascending order of ordinals starting from 0 and ending with ω^ω. Here we will continue this train of thought.

Definition 2.25 (epsilon number)**.** Let α be an ordinal; α is called an *epsilon number* if $\alpha = \omega^\alpha$.

It follows trivially that α is neither zero nor a successor ordinal, and is therefore a limit ordinal. We can use this intuition and the transfinite recursion theorem to define the first epsilon number, for which we'll need the following lemma.

Lemma 2.9. Let $\{\beta_i \mid i \in I\}$ be a set of ordinals, and let $\beta = sup(\{\beta_i \mid i \in I\})$. Then $\alpha^\beta = sup(\{\alpha^{\beta_i} \mid i \in I\})$.

Theorem 2.10 (ϵ_0)**.** Using the transfinite recursion theorem, define $f : \omega \to OR$ such that

$$f(0) = 1$$
$$f(n+1) = \omega^{f(n)}.$$

As an illustration, this means $f(0) = 1, f(1) = \omega, f(2) = \omega^\omega, f(3) = \omega^{\omega^\omega}$, and so forth.

We may define ϵ_0 as $sup(\{f(\beta) \mid \beta < \omega\})$. Then

$$\begin{aligned}
\omega^{\epsilon_0} &= sup(\{\omega^{f(\beta)} \mid \beta < \omega\}) \quad \text{(By Lemma 2.9)} \\
&= sup(\{f(\beta+1) \mid \beta < \omega\}) \\
&= sup(\{f(\beta) \mid \beta < \omega\}) \quad (\text{As } \beta < \beta+1 < \omega) \\
&= \epsilon_0.
\end{aligned}$$

And thus ϵ_0 is an epsilon number. In fact, it is the least such epsilon number.

The idea here should seem familiar; we've constructed some ascending well order and asked for its supremum. This is the same mechanism by which we built ω. Since $\epsilon_0 = \omega^{\epsilon_0}$ is a recursive definition, we can also visualize it, expanded, as $\omega^{\omega^{\omega^{\cdots}}}$.

By the same logic applied to ω, it also follows that $\epsilon_0 < \epsilon_0 + 1 < ...$, and so forth. And continually, we may also build successively larger epsilon numbers via the same construction as we built ϵ_0 (just generalize theorem 2.10 to f_α such that $f_\alpha(0) = \alpha + 1$ and $f_\alpha(n+1) = \omega^{f_\alpha(n)}$).

3 An Encoding in λ-Calculi

3.1 The untyped λ-calculus

In *Formal definitions in the theory of the ordinal numbers*, Church and Kleene wished to extend the functional encoding of the natural numbers (which we observed in class) to the ordinal numbers [CK37].

They thus defined an encoding as follows, where $1, 2, 3, ...$ represent the corresponding Church encoded naturals[2].

[2] I've adapted this untyped encoding to Cedille at `https://github.com/ahubers/cedille-ordinals/blob/master/untyped-church-encoding/Untyped-Ord.ced`, for those interested.

$$
\begin{aligned}
0_o &= \lambda\, m.m\, 1 && \text{(The ordinal zero)}\\
S_o &= \lambda\, a.\, \lambda\, m.\, (m\, 2)\, a && \text{(the Successor of an ordinal } a\text{)}\\
L &= \lambda\, a.\, \lambda\, r.\, \lambda\, m.\, ((m\, 3)\, a)\, r && \text{(the Limit of ordinal } a\text{)}\\
1_o &= S_o\, 0_o\\
2_o &= S_o\, 1_o\\
&\ldots
\end{aligned}
$$

They then prescribe the following rules to make sense of these formulae (rewritten here in the notation we used in class in place of the older notation from the work cited):

1. If a' represents the ordinal a and $b \rightsquigarrow^* a'$, then b also represents a.
2. 0_o represents the ordinal zero.
3. If a' represents the ordinal a, then $S_o(a')$ represents the successor of a.
4. If b is the limit (i.e, supremum) of an increasing sequence of ordinals $b_0, b_1, b_2, ...$, and if r is a formula such that the formulas $r(0_o), r(1_o), r(2_o), ...$ represents the ordinals $b_0, b_1, b_2, ...$, respectively, then $(L\, 0_o)\, r$ represents b.

Rules 1 through 3 should be fairly obvious. The fourth rule is where we deviate from the naturals to be able to define limit ordinals.

We can dissect this rule as stating, if I have some function r that takes all of the finite ordinals and maps them to an increasing sequence of ordinals, then I can give 0_o and r to L in order to get the supremum of that increasing sequence.

We can immediately apply this rule to obtain ω. Recall that ω is the supremum of the strictly increasing sequence $0_o, 1_o, 2_o, ...$; Consequently, $\omega = (L\, 0_o)(\lambda x.\, x)$.

This idea is what gives us the intuition about how to extend from the naturals to the ordinals as an encoding. You can rephrase it equivalently as saying, let r map finite ordinals (aka naturals) to ordinals. Then pass r to L and you get the supremum of r. Consequently we now have a way in which we can go beyond the naturals.

3.2 System F

Our Typing in System F will not type the terms provided by Kleene and Church above, but it will be derived from the same inspiration.

You can encode the naturals in System F as follows:

$$Nat := \forall X.\, X \to (X \to X) \to X$$

Breaking this apart, we can think of the first X as the "zero" case, and the $(X \to X)$ segment as the "step" case. Inductively, we could also define a Nat as two constructors:

$$Zero :: Nat$$
$$Succ :: Nat \to Nat$$

The extension to the ordinals follows from adding in our limit case ($(Nat \to X) \to X$):

$$Ord := \forall X.\, X \to (X \to X) \to \underline{((Nat \to X) \to X)} \to X$$

Where similarly we get the constructors:

$$Zero_o :: Ord$$
$$Succ_o :: Ord \to Ord$$
$$Limit_o :: (Nat \to Ord) \to Ord$$

You can think of the Limit constructor as saying "Give me a strictly ascending sequence of Ord with a least element, represented as the function typed by $Nat \to Ord$, and I'll give you back an Ord".

Given our definition of Ord, we can define each of its constructors as terms in System F as follows:

$$Zero_o := \lambda\, z.\, \lambda\, s.\, \lambda\, l.\, z$$
$$Succ_o := \lambda\, a.\, \lambda\, z.\, \lambda\, s.\, \lambda\, l.\, s\, (a\, z\, s\, l)$$
$$Limit_o := \lambda\, f.\, \lambda\, z.\, \lambda\, s.\, \lambda\, l.\, l\, (\lambda\, n.\, f\, n\, z\, s\, l).$$

4 Encoding the Ordinals in Cedille up to ϵ_0

Cedille is an interactive theorem-prover and dependently typed programming language, based on extrinsic (aka Curry-style) type theory, and developed here at the University of Iowa. By "extrinsic" or "Curry-style" type theory, we mean that a type is an annotation for a term – there's nothing about the type that embeds itself intrinsically into the term. So a term, say $\lambda x.x$ can (and does) have multiple types within Cedille.

Cedille has a number of exotic language features, however I'm going to restrict our usage to just a syntax and implementation that should look familiar. Namely, I won't need to make use of any dependent typing. In fact, for the basic encoding and definitions, the implementation should feel fairly correspondent to the System F description given in Section 3.

One novel feature worth noting, however, is that Cedille is impredicative. This means when you see a $\forall X.T$ for some type T, it means we are permitted to let X be instantiated with $\forall X.T$. This is true of System F as well, and we will use this fact to simplify Escardo's development.

4.1 A First Implementation

We're first going to follow with a translation of how Escardo proceeds in Agda [Esc11], which begins with a higher kinded definition of the type `Ord` as:

```
Ord : Set → Set
Ord X = X → (X → X) → ((ℕ → X) → X) → X
```

Of note is that Agda has a type hierarchy universe – so `Set` (which is shorthand for `Set1`) is the first type of all types, and `Set2` is the type of `Set1`, and so forth[3]. As a consequence, the type operator Ord may only accept types of type `Set` (but not of type `Set2`!).

To contrast, when we translate this definition to Cedille, we need only specify the kind of `X`, where you may think of "kinds" as being the "type signature" of type operators. For example, `Ord : ⋆ → ⋆` means "I take in a type of kind ⋆ and return a type of kind ⋆." ⋆ is the kind of "concrete" types – meaning they aren't expecting to be given any additional type parameters.

```
Ord : ⋆ → ⋆ = λ X: ⋆. X → (X → X) → ((Nat → X) → X) → X.
```

And accordingly, we've said that Ord is higher kinded – it needs you to give it some concrete type `X` so that it may define `Ord ·X`. We can now build our constructors using the same terms as defined in Section 3.2.

```
zer : ∀ X: ⋆. Ord ·X
    = Λ X. λ z. λ s. λ l. z.

suc : ∀ X: ⋆. Ord ·X → Ord ·X
    = Λ X. λ a. λ z. λ s. λ l. s (a z s l).

lim : ∀ X: ⋆. (Nat → Ord ·X) → Ord ·X
    = Λ X. λ f. λ z. λ s. λ l. l (λ i. f i z s l).
```

Here I should point out that Cedille makes use of the the symbol Λ to give the quantified type `X` as argument to the term. For example, the term for zero is `Λ X. λ z. λ s. λ l. z`, where `Λ X` simply allows us to reference the type `X`. Sometimes this is not needed, but sometimes we do need it.

Additionally, the type operator `Ord` needs a type `X` of kind ⋆, so we quantify over all concrete `X`. The symbol · in Cedille indicates type application: Ord needs a type, so `Ord ·X` says "give X to Ord as type variable". Consequently, `Ord ·X` has kind ⋆.

[3]For a more comprehensive explanation, see `https://agda.readthedocs.io/en/v2.6.1.1/language/universe-levels.html`, which I encourage you to read, if not for then this report then just because it's quite interesting.

At this point, Cedille type checks – meaning our terms really do have the types we said they would! The next logical step to take is that we would like to define some operations upon our type. We have such an interface in the form of the ordinal arithmetic defined in Section 2.6.

We can first define add as follows:

```
ordAdd : ∀ X: ⋆. Ord ·X → Ord ·X → Ord ·X
       = Λ X. λ a. λ b. λ z. λ s. λ l. a (b z s l) s l.
```

Informally, what we've done is supplied the `X` obtained from the ordinal application `b z s l` as the "base case" of the ordinal `a`.

As a sanity check, we can employ some of Cedille's tooling to verify this is doing what we want.

```
one : ∀ X: ⋆. Ord ·X = Λ X. suc (zer ·X).
sanityCheck : { ordAdd one ≃ suc } = β.
```

Here I've defined `one` as the successor of zero, and assigned the type `{ ordAdd one ≃ suc }` to the term `sanityCheck`. Then by inhabiting the term with the symbol β, we are asking Cedille if the left hand side and right hand side of the equality type `{ ordAdd one ≃ suc }` are actually beta-eta equivalent terms, which Cedille type-checks and concludes they are. Consequently, the function that adds `one` is equivalent to our definition of the ordinal successor, which is of course desired!

In order to continue with defining multiplication, we would like to make use of the fact that multiplication is in fact just repeated addition. In other words, multiplication can be thought of as addition applied recursively. To express this logically, we can define iterative recursion over any `Ord` and `X` as follows:

```
o-rec : ∀ X: ⋆. X → (X → X) → ((Nat → X) → X) → Ord ·X → X
      = Λ X. λ z. λ s. λ l. λ o. o z s l.
```

You can think of this as "take the the ordinal `o` and give it a new base, step, and limit case." For example, suppose we wanted just the even finite ordinals:

```
evenFiniteOrdinals :  ∀ X: ⋆. Ord ·(Ord ·X) → Ord ·X
            =  Λ X. o-rec (zer ·X) (λ n. suc (suc n)) (lim ·X).
```

This term says "start with zero, and in the successor case, take the successor twice instead of just once (so that the result is 2, 4, 6, ..., and so forth)".

With this definition we can then define multiplication as the term:

```
λ a. o-rec (zer ·X) (λ r. ordAdd r a) (lim ·X).
```

However, here we encounter a problem with the encoding. We would like multiplication, like addition, to be of type `Ord → Ord → Ord`. However this term does not inhabit that type! In fact its actual type is:

```
∀ X: ⋆. Ord ·X → Ord ·(Ord ·X) → Ord ·X
```

E.g, we define ordinal multiplication as

```
ordMul : ∀ X: ⋆. Ord ·X → Ord ·(Ord ·X) → Ord ·X
       = Λ X. λ a. o-rec (zer ·X) (λ r. ordAdd r a) (lim ·X).
```

Which is to say, in the definition of `o-rec`, we are instantiating the type variable `X` with `Ord ·X`. This is the same case we observed in our even finite ordinal example. You can think of `o-rec` really as a function which take a base, step, and limit case, and returns a function of type `Ord ·X → X`. Naturally, letting `X` be substituted in with `Ord ·X` gives us the type `Ord ·(Ord ·X) → Ord ·X`. As a consequence, our typing of multiplication is not uniform with respect to addition.

The same applies for exponentiation:

```
ordExp : ∀ X: ⋆. Ord ·(Ord ·X) → Ord ·(Ord ·X) → Ord ·X
       = Λ X. λ a. o-rec (suc (zer ·X)) (λ r. ordMul r a) (lim ·X).
```

Ideally we would like our ordinal arithmetic operators to be uniform so that we may apply them recursively to obtain limit ordinals.

Before preceding, let's define a combinator `rec` that, given a base case and a step case, each of some arbitrary type `X`, will produce for you a `Nat → X` function.

```
rec : ∀ X: ⋆. X → (X → X) → Nat → X =
    Λ X. λ x. λ f. λ n.
    μ rec'. n {
      | zero → x
      | succ n' → f (rec' n')
}.
```

Even without explaining the syntax and mechanics of the code above, this concept should feel familiar: this is what we did when trying to build ϵ_0. We define a base and a step case, and then let the function obtained from `rec` return the base when its input is zero, and return the step case applied recursively when its input is a successor.

We may actually attempt to build ω in our encoding with the same idea.

```
finiteOrdinals : ∀ X: ⋆. Nat → Ord ·X = Λ X. rec (zer ·X) (suc ·X).
omega : ∀ X: ⋆. Ord ·X = Λ X. lim (finiteOrdinals ·X).
```

Here we build ω by saying it's the limit of the finite ordinals. the finite ordinals are obtained by the combinator `rec` with base case zero and step case successor.

At this point we note that to build `finiteOrdinals`, the successor step must have type `Ord ·X → Ord ·X` to work. This is true when defining the finite ordinal sequence, but what if we want to build ϵ_0 by defining the sequence $\omega, \omega^\omega, \omega^{\omega^\omega}, ...$?

```
omegaTower : ∀ X: ⋆. Nat → Ord ·X
           = Λ X. rec (omega ·X) ((ordExp ·X) (omega ·X)).
```

Here Cedille does **not** type check, as our successor case `((ordExp ·X) (omega ·X))` has type `Ord ·(Ord ·X) → Ord ·X`. Consequently, it is not permitted as an argument to `rec`.

So how do we get around this? In Agda, Escardo uses some clever manipulation via dependent typing and an ascending hierarchy of successively bigger types to make the typing of multiplication and exponentiation uniform. However in Cedille we can utilize the power of impredicativity to redefine our encoding so that this problem basically vanishes.

4.2 Reimplementing with Impredicativity

Let's redefine our `Ord` encoding in Cedille, but this time with `∀ X` instead of `λ X: ⋆`. This minor change has drastic connotations – namely that, in Cedille, we're perfectly allowed to let `X` be instantiated with `Ord` itself.

```
Ord : ⋆ = ∀ X: ⋆. X → (X → X) → ((Nat → X) → X) → X.
```

Continuing as before, we can redefine our constructors as well as the terms `o-rec` and `ordAdd`, and Cedille will type check to assert they are in fact correct.

```
zer : Ord
    = Λ X. λ z. λ s. λ l. z.

suc : Ord → Ord
    = λ a. Λ X. λ z. λ s. λ l. s (a z s l).

lim : (Nat → Ord) → Ord
    = λ f. Λ X. λ z. λ s. λ l. l (λ n. f n z s l).

ordAdd : Ord → Ord → Ord
       = λ a. λ b. Λ X. λ z. λ s. λ l. a (b z s l) s l.
```

At this point in the previous section, we got stuck trying to define ordinal multiplication with the type `Ord → Ord → Ord`.

Let's try it now:

```
ordMul : Ord → Ord → Ord
       = λ a. o-rec (zer) (λ b. ordAdd b a) lim.
```

This type checks without a problem! Why is this?

Let's examine the successor case `(λ b. ordAdd b a)`. As desired, it has type `Ord → Ord`. How has impredicativity permitted us to do this? Recall that o-rec has type `∀ X: ⋆. X → (X → X) → ((Nat → X) → X) → Ord → X`, and that we can think of its "return type" really as `Ord → X`. In our previous encoding, we had to let `X` be instantiated with `Ord ·X` for some concrete type `X`. Here we can simply let `X` be `Ord`, and we're returned a function with type `Ord → Ord`.

```
o-rec :∀ X: ⋆. X → (X → X) → ((Nat → X) → X) → Ord → X
      = Λ X. λ z. λ s. λ l. λ o. o z s l.
```

Looking at `o-rec` as it's used in `ordMul`, we see that bound terms `z`, `s`, and `l` are of type `Ord`, `(Ord → Ord)`, and `((Nat → Ord) → Ord)`, respectively. We then pass them to the ordinal `o` as base, step, and limit case. However `o`, being an ordinal, has type `∀ X: ⋆. X → (X → X) → ((Nat → X) → X) → X`. So we are passing to the term `o` terms of type `Ord`. As stated before, our encoding of `Ord` lets `X` be `Ord` itself. This is the power of impredicativity – `Ord` is defined in terms of **all concrete types**, including itself!

We can obtain a similar result for exponentiation:

```
ordExp : Ord → Ord → Ord
       = λ a. o-rec (suc zer) (λ b. ordMul b a) lim.
```

Now that exponentiation has the desired type, we may use it to construct ϵ_0:

```
finiteOrdinals : Nat → Ord = rec zer suc.
omega : Ord = lim finiteOrdinals.


omegaTower : Nat → Ord
= rec omega (ordExp omega).

epsilon0 : Ord = lim omegaTower.
```

Where we differ from before is the successor step `(ordExp omega)` passed to `rec` has the correct type `Ord → Ord`. Thereafter we just take the limit of the `omegaTower` and get ϵ_0.

4.3 Further Work

What I've covered so far in this report is just the base implementations of an ordinal encoding in Cedille. There's much more you can do with it. I've supplied the source at `https://www.github.com/ahubers/cedille-ordinals/`. In this repo you can find additional explorations, such as:

- An implementation of our encoding using Cedille's derived inductive types (see `typed-encoding/Ordinal-03.ced`).
- Equality as a relational type over the inductive datatype (see `typed-encoding/OrdEq.ced`).
- Some basic theorems over the ordinals using the relational type `OrdEq`.

The hope is that this report should provide more than sufficient grounds for you to comprehend the rest of this work. Additionally, in the repository's readme I provide a list of topics which I didn't cover, but you would be able to explore if interested.

References

[CK37] Alonzo Church and S. C. Kleene. Formal definitions in the theory of ordinary numbers. *Fundamenta Mathematicae*, pages 11–21, 1937.

[Esc11] Martin Escardo. Ordinals in goedel's system t and in martin-loef type theory. `https://www.cs.bham.ac.uk/~mhe/papers/ordinals/ordinals.html`, 2011.

[Jec06] Thomas Jech. *Set Theory, Third Millenium Edition.* Springer, Berlin, 2006.

[JJ98] Gareth A. Jones and J. Mary Jones. *Elementary Number Theory.* Springer, London, 1998.

[Pin14] Charles C. Pinter. *A Book of Set Theory.* Dover Publications, Mineola, New York, 2014.