A Computational Approach to the Ordinal Numbers

Documents ordCalc_0.2

Paul Budnik
Mountain Math Software
paul@mtnmath.com

Contents

List of Tables

1 Introduction

The ordinal calculator is a tool for learning about the ordinal hierarchy and ordinal notations. It is also a research tool. Its motivating goal is ultimately to expand the foundations of mathematics by using computer technology to manage the combinatorial explosion in complexity that comes with explicitly defining the recursive ordinals implicitly defined by the axioms of Zermelo Frankel set theory[4, 2]. The underlying philosophy focuses on what formal systems tell us about physically realizable combinatorial processes.[3].

Appendix A "Using the ordinal calculator" is the user's manual for the interactive mode of this program. It describes how to download the program and use it from the command line. It is available as a separate manual at: `www.mtnmath.com/ord/ordCalc.pdf`. This document is intended for those who want to understand the theory on which the program is based, understand the structure of the program or use and expand the program in C++.

All the source code and documentation (including this manual) is licensed for use and distribution under the GNU General Public License, version 2.

1.1 Intended audience

This document is targeted to mathematicians with limited experience in computer programming and computer scientists with limited knowledge of the foundations of mathematics. Thus it contains substantial tutorial material, often as footnotes. The ideas in this paper have been implemented in the C++ programming language. C++ keywords and constructs defined in this program are in `teletype font`. This paper is both a top level introduction to this computer code and a description of the theory that the code is based on. The C++ tutorial material in this paper is intended to make the paper self contained for someone not familiar with the language. However it is not intended as programming tutorial. Anyone unfamiliar with C++, who wants to modify the code in a significant way, should consult one of the many tutorial texts on the language. By using the command line interface described in Appendix A, one can use many of the facilities of the program interactively with no knowledge of C++.

1.2 The Ordinals

The ordinals are the backbone of mathematics. They generalize induction on the integers[1] in an open ended way. More powerful modes of induction are defined by defining larger ordinals and not by creating new laws of induction.

The smallest ordinals are the integers. Other ordinals are defined as infinite sets[2]. The smallest infinite ordinal is the set of all integers. Infinite objects are not subject to computational manipulation. However the set of all integers can be represented by a computer program that lists the integers. This is an abstraction. Real programs cannot run forever error free. However the program itself is a finite object, a set of instructions, that a computer program can manipulate and transform. Ordinals at or beyond ω may or may not exist as infinite objects in some ideal abstract reality, but many of them can have their structure represented by a computer program. This does not extend to ordinals that are not countable, but it can extend beyond the recursive ordinals[3].

[1]Induction on the integers states that a property holds for every integer $n \geq 0$, if it is true of 0 and if, for any integer x, if it is true for x, it must be true for $x + 1$.
$[p(0) \wedge \forall_{x \in N} p(x) \rightarrow p(x+1)] \rightarrow \forall_{x \in N} p(x)$

[2]In set theory 0 is the empty set. 1 is the set containing the empty set. 2 is the set containing 0 and 1. Each finite integer is the union of all smaller integers. Infinite ordinals are also constructed by taking the union of *all* smaller ones. There are three types of Ordinals. 0 or the empty set is the smallest ordinal and the only one not defined by operations on previously defined ordinals. The successor to an ordinal a is the union of a and the members of a. In addition to 0 and successor ordinals, there are limit ordinals. The smallest limit ordinal is the set of all integers or all finite successors of 0 called ω.

Ordinals are well ordered by the relation of set membership, $\in$. For any two ordinals a and b either $a \in b$, $b \in a$ or $a = b$.

A limit ordinal consists of an infinite collection of ordinals that has no maximal or largest element. For example there is no single largest integer. Adding one to the largest that has been defined creates a larger integer.

[3]The recursive ordinals are those whose structure can be fully enumerated by an ideal computer program that runs forever. For any recursive ordinal a there exists a computer program that can enumerate a unique symbol or notation for every ordinal less than a. For any two notations for ordinals less than a, there is recursive algorithm that can determine the relative ranking (less than, equal to or greater than) of the two ordinals. Larger countable ordinals cannot have their structure fully enumerated by a recursive process. but

Ordinal notations assign unique finite strings to a subset of the countable ordinals. Associated with a notation system is a recursive algorithm to rank ordinal notations ($<$, $>$ and $=$). For recursive ordinals there is also an algorithm that, given an input notation for some ordinal α, enumerates notations of all smaller ordinals. This latter algorithm cannot exist for ordinals that are not recursive but an incomplete variant of it can be defined for countable ordinals.

The ultimate goal of this research is to construct notations for large recursive ordinals eventually leading to and beyond a recursive ordinal that captures the combinatorial strength of Zermelo-Frankel set theory (ZF[4]) and thus is strong enough to prove its consistency. Computers can help to deal with the inevitable complexity of strong systems of notations. They allow experiments to tests ones intuition. Thus a system of notations implemented in a computer program may be able to progress significantly beyond what is possible with pencil and paper alone.

2 Ordinal notations

The ordinals whose structure can be enumerated by an ideal computer program are called recursive. The smallest ordinal that is not recursive is the Church-Kleene ordinal ω_1^{CK}. For simplicity this is written as ω_1[5]. Here the focus is on notations for recursive ordinals. The set that defines a specific ordinal in set theory is unique, but there are many different computer programs that can enumerate the structure of the same recursive ordinal. A system of recursive representations or notations for recursive ordinals, must recursively determine the relative size of any two notations. This is best done with a unique notation or normal form for each ordinal represented. Thus an ordinal notation system should satisfy the following requirements:

1. There is a unique finite string of symbols that represents every ordinal within the system. These are ordinal notations.

2. There is an algorithm (or computer program) that can determine for every two ordinal notations a and b if $a < b$ or $a > b$ or $a = b$[6]. One and only one of these three must hold for every pair of ordinal notations in the system.

3. There is an algorithm that, given an ordinal notation a as input, will output an infinite sequence of ordinal notations, $b_i < a$ for all integers i. These outputs must satisfy the

they can be defined as properties of recursive processes that operate on a partially enumerable domain. For more about this see Section 8.

[4] ZF is the widely used Zermelo-Frankel formulation of set theory. It can be thought of as a one page computer program for enumerating theorems. See either of the references [4, 2] for these axioms. Writing programs that define notations for the recursive ordinals definable in ZF can be thought of as an attempt to make explicit the combinatorial structures implicitly defined in ZF.

[5] ω_1 is most commonly used to represent the ordinal of the countable ordinals (the smallest ordinal that is not countable). Since this paper does not deal with uncountable sets (accept indirectly in describing an existing approach to ordinal collapsing in Section 8.4) we can simplify the notation for ω_1^{CK} to ω_1.

[6] In set theory the relative size of two ordinals is determined by which is a member, $\in$, of the other. Because notations must be finite strings, this will not work in a computational approach. An explicit algorithm is used to rank the size of notations.

property that eventually every ordinal notation $c < a$ will be output if we recursively apply this algorithm to a and to every notation the algorithm outputs either directly or indirectly.

4. Each ordinal notation must represent a unique ordinal as defined in set theory. The union of the ordinals represented by the notations output by the algorithm defined in the previous item must be equal to the ordinal represented by a.

By generalizing induction on the integers, the ordinals are central to the power of mathematics[7]. The larger the recursive ordinals that are provably definable within a system the more powerful it is, at least in terms of its ability to solve consistency questions.

Set theoretical approaches to the ordinals can mask the combinatorial structure that the ordinals implicitly define. This can be a big advantage in simplifying proofs, but it is only through the explicit development of that combinatorial structure that one can fully understand the ordinals. That understanding may be crucial to expanding the ordinal hierarchy. Beyond a certain point this is not possible without using computers as a research tool.

2.1 Ordinal functions and fixed points

Notations for ordinals are usually defined with strictly increasing ordinal functions on the countable ordinals. A fixed point of a function f is an ordinal a with $f(a) = a$. For example $f(x) = n + x$ has every limit ordinal as a fixed point provided n is an integer. The Veblen hierarchy of ordinal notations is constructed by starting with the function ω^x. Using this function, new functions are defined as a sequence of fixed points of previous functions[15, 11, 6]. The first of these functions, $\varphi(1, \alpha)$ enumerates the fixed points of ω^α. $\varphi(1, \alpha) = \varepsilon_\alpha$.

The range and domain of these functions are an expandable collection of ordinal notations defined by `C++ class`, `Ordinal`. The computational analog of fixed points in set theory involves the extension of an existing notation system. A fixed point can be thought of as representing the union of all notations that can be obtained by finite combinations of existing operations on existing ordinal notations. To represent this fixed point ordinal, the notation system must be expanded to include a new symbol for this ordinal. In addition the algorithms that operate on notations must be expanded to handle the additional symbol. In particular the recursive process that satisfies item 3 on page 6 must be expanded. The goal is to do more than add a single new symbol for a single fixed point. The idea is to define powerful expansions that add a rich hierarchy of symbolic representations of larger ordinals.

The simplest example of a fixed point is ω the ordinal for the integers. It cannot be reached by any finite sequence of integer additions. Starting with a finite integer and adding

[7] To prove a property is true for some ordinal, a, one must prove the following.

1. It is true of 0.
2. If it is true for any ordinal $b < a$ it must be true of the successor of b or $b + 1$.
3. If it is true for a sequence of ordinals c_i such that $\bigcup_i c_i = c$ and $c \leq a$, then it is true of c.

ω to it cannot get past ω. $n + \omega = \omega$ for all finite integers n.[8] The first fixed point for the function, ω^x, is the ordinal $\varepsilon_0 = \omega + \omega^\omega + \omega^{(\omega^\omega)} + \omega^{(\omega^{(\omega^\omega)})} + ...$ (The parenthesis in this equation are included to make the order of the exponentiation operations clear. They will sometimes be omitted and assumed implicitly from now on.) ε_0 represents the union of the ordinals represented by notations that can be obtained from finite sequences of operations starting with notations for the ordinals 0 and ω and the ordinal notation operations of successor (+1), addition, multiplication and exponentiation.

2.2 Beyond the recursive ordinals

The `class Ordinal` is not restricted to notations for recursive ordinals. Admissible ordinals extend the concept of recursive ordinals by considering ordinal notations defined by Turing Machines (TM) with oracles[9][9]. There is an alternative way to extend the idea of recursive notations for ordinals that does not involve the completed infinite set required to define an oracle. It is more appropriate for a computational approach to the ordinals. The recursive ordinals can be characterized by recursive processes that map integers to either processes like themselves or integers. That is a computer program that accepts an integer as input and outputs either a computer program like itself or an integer.

For such a computer program to represent the structure of an ordinal it must be well founded[10]. This means, if one applies any infinite sequence of integer inputs to the base program, it will terminate[11] Of course it must meet all the other requirements for a notation system.

A recursive process satisfying the requirements on page 6 is well founded and represents the structure of a recursive ordinal. It has been shown that the concept of recursive process well founded for infinite sequences of integers can fully characterize the recursive ordinals[13]. One can generalize this idea to recursive processes well founded for an infinite sequences of notations for recursive ordinals. And of course one can iterate this definition in simple and complex ways. In this way one can define countable ordinals $> \omega_1^{CK}$ as properties of recursive processes. In contrast to recursive ordinal notations, notations for larger ordinals cannot be associated with algorithms that fully enumerate the structure of the ordinal they represent. However it is possible to define a recursive function that in some ways serves as a substitute (see Section 8.1 on `limitOrd`).

[8]A fixed point for addition is called a *principal additive ordinal.* Any ordinal $a > 0$ such that $a + b = b$ for all $a < b$ is an additive principle ordinal.

[9]A TM oracle is an external device that a TM can query to answer questions that are recursively unsolvable like the computer halting problem. One can assume the existence of an infinite set (not recursive or recursively enumerable) that defines a notation system for all recursive ordinals and consider what further notations are definable by a recursive process with access to this oracle.

[10]In mathematics a well founded relationship is one with no infinite descending chains. Any collection of objects ordered by the relationship must have a minimal element.

[11]The formulation of this property requires quantification over the reals and thus is considered impredicative. (Impredicative sets have definitions that assume their own existence. Some reals can only be defined by quantifying over the set of all reals and this makes them questionable for some mathematicians.) In a computational approach one need not assume there is a set of all objects satisfying the property. Instead one can regard it as a computationally useful property and build objects that satisfy it in an expanding hierarchy. Impredictivity is replaced with explicit incompleteness.

2.3 Uncountable ordinals

The ordinal of the countable ordinals, Ω, cannot have a computational interpretation in the sense that term is used here. Uncountable ordinals exist in a through the looking glass reality. It is consistent to argue about them because mathematics will always be incomplete. The reals *provably definable* within a formal system form a definite totality. The formulas that define them are recursively enumerable. They are uncountable *from within* the system that defines them but they are countable when viewed externally.

Building incompleteness into the system from the ground up in lieu of claiming cardinality has an *absolute* meaning will, I believe, lead to a more powerful and more understandable mathematics. That is part of the reason I suspect a computational approach may extend the ordinal hierarchy significantly beyond what is possible by conventional proofs. Writing programs that define ordinal notations and operations on them provides powerful tools to help deal with combinatorial complexity that may vastly exceed the limits of what can be pursued with unaided human intelligence. This is true in most scientific fields and there is no reason to think that the foundations of mathematics is an exception. The core of this paper is C++ code that defines a notation system for an initial fragment of the recursive ordinals and a subset of larger countable ordinals.

The development of this computational approach initially parallels the conventional approach through the Veblen hierarchy (described in Section 5). The C++ `class Ordinal` is incompletely specified. C++ subclasses and `virtual` functions allow a continued expansion of the `class` as described in the next section. The structure of the recursive ordinals defined in the process are fully specified. Ordinals $\geq \omega_1^{CK}$ are partially specified.

Mathematics is an inherently creative activity. God did not create the integers, they like every other infinite set, are a human conceptual creation designed to abstract and generalize the real finite operations that God, or at least the forces of nature, did create. The anthology, *God Created the Integers*[7], collects the major papers in the history of mathematics. From these and the commentary it becomes clear that accepting *any infinite totalities* easily leads one to something like ZF. If the integers are a completed totality, then the rationals are ordered pairs of integers with 1 as their only common denominator. Reals are partitions of the rationals into those less than a given real and those greater than that real. This is the Dedekind cut. With the acceptance of that, one is well on the way to the power set axiom and wondering about the continuum hypothesis[12]. This mathematics is not false or even irrelevant, but it is only meaningful relative to a particular formal system. Thinking such questions are absolute leads one down a primrose path of pursuing as absolute what is not and cannot be objective truth beyond the confines of a particular formal system.

[12]The continuum hypothesis is the assertion that the reals have the smallest cardinality greater than the cardinality of the integers. This means there is no set that the integers cannot be mapped onto and that cannot be mapped onto the reals.

3 Program structure

The C++ programming language[13] has two related features, subclasses and `virtual` functions that are useful in developing ordinal notations. The base `class`, `Ordinal`, is an open ended programming structure that can be expanded with subclasses. It does not represent a specific set of ordinals and it is not limited to notations for recursive ordinals. Any function that takes `Ordinal` as an argument must allow any subclass of `Ordinal` to be passed to it as an argument. The reverse does not hold.

3.1 `virtual` functions and subclasses

Virtual functions facilitate the expansion of the base `class`. For example there is a base `class virtual` member function `compare` that returns 1, 0, or -1 if its argument (which must be an element of `class Ordinal`) is less than, equal to or greater than the ordinal notation it is a member function of. The base `class` defines notations for ordinals less than ε_0. As the base `class` is expanded, an extension of the `compare virtual` function must be written to take care of cases involving ordinals greater than ε_0. Programs that call the `compare` function will continue to work properly. The correct version of `compare` will automatically be invoked depending on the type of the object (ordinal notation) from which `compare` is called. The original `compare` does not need to be changed but it does need to be written with the expansion in mind. If the argument to `compare` is not in the base `class` then the expanded function must be called. This can be done by calling `compare` recursively using the argument to the original call to `compare` as the `class` instance from which `compare` is called.

It may sound confusing to speak of subclasses as expanding a definition. The idea is that the base `class` is the broadest `class` including all subclasses defined now or that might be defined in the future. The subclass expands the objects in the base `class` by defining a limited subset of new base `class` objects that are the only members of the subclass (until and unless it gets expanded by its own subclasses). This is one way of dealing with the inherent incompleteness of a computational approach to the ordinals.

3.2 Ordinal normal forms

Crucial to developing ordinal notations is to construct a *unique* representation for every ordinal. The starting point for this is the Cantor normal form. Every ordinal, α, can be represented by an expression of the following form:

$$\alpha = \omega^{\alpha_1} n_1 + \omega^{\alpha_2} n_2 + \omega^{\alpha_3} n_3 + ... + \omega^{\alpha_k} n_k \tag{1}$$

$$\alpha_1 > \alpha_2 > \alpha_3 > ... > \alpha_k$$

[13]C++ is an object oriented language combining functions and data in a `class` definition. The data and code that form the `class` are referred to as `class` members. Calls to non`static` member functions can only be made from an instance of the `class`. Data `class` members in a member function definition refer to the particular instance of the `class` from which the function is called.

The α_k are ordinals and the n_k are integers > 0.

Because $\varepsilon_0 = \omega^{\varepsilon_0}$ the Cantor normal form gives unique representation only for ordinals less than ε_0. This is handled by requiring that the normal form notation for a fixed point ordinal be the simplest expression that represents the ordinal. For example, A notation for the Veblen hierarchy used in this paper (see Section 5) defines ε_0 as $\varphi(1,0)$. Thus the normal form for ε_0 would be $\varphi(1,0)$[14] not $\omega^{\varphi(1,0)}$. However $\varphi(1,0)^2$ is displayed as $\omega^{\varphi(1,0)\times 2}$.

The `Ordinal` base `class` represents an ordinal as a linked list of terms of the form $\omega^{\alpha_k}n_k$. This limits the unexpanded base `class` to ordinals of the form ω^α where α is a previously defined member of the base `class`. These are the ordinals less than ε_0. The base `class` for each term of an `Ordinal` is `CantorNormalElement`.

3.3 Memory management

Most infinite `Ordinals` are constructed from previously defined `Ordinals`. Those constructions need to reference the ordinals used in defining them. The `Ordinals` used in the definition of another `Ordinal` must not be deleted while the object that uses them still exists. This often requires that `Ordinals` be created using the `new` C++ construct. Objects declared without using `new` are automatically deleted when the block in which they occur exits.

This program does not currently implement any garbage collection. Declaring many `Ordinals` using `new` will eventually exhaust available memory.

4 `Ordinal` base `class`

The base `class Ordinal`[15] includes all ordinals notations defined now or in the future in this system. This `class` with no sub`class`es defines notations for the ordinals $< \varepsilon_0$. Sections 6, 7 and 9 describe three sub`class`es that extend the notation system to large recursive ordinals and to admissible countable ordinals.

The finite (integer) `Ordinals` and ω, the ordinal that contains all integers, are defined using the constructor[16] for `class Ordinal.`[17]. Other base `class` ordinals are usually defined using expressions involving addition, multiplication and exponentiation (see Section 4.4) of previously defined `Ordinals`.

The `Ordinal` for 12 is written as "`const Ordinal& twelve = * new Ordinal(12)`"[18] in C++. The `Ordinals` for `zero`, `one` and `omega` are defined in global name space `ord`[19] The

[14]In the ordinal calculator $\varphi(1,\alpha)$ is displayed as ε_α.

[15]'`Ordinal`' when capitalized and in `tty font`, refers to the expandable C++ `class` of ordinal notations.

[16]The constructor of a `class` object is a special member function that creates an instance of the `class` based on the parameters to the function.

[17]The integer ordinals are not defined in this program using the C++ `int` data type but a locally defined `Int` data type that uses the Gnu Multiple Precision Arithmetic Library to allow for arbitrarily large integers depending on the memory of the computer the program is run on.

[18]In the interactive ordinal calculator (see Appendix A) write "`twelve = 12`" or just use `12` in an expression.

[19]All global variables in this implementation are defined in the `namespace ord`. This simplifies integration with existing programs. Such variables must have the prefix '`ord::`' prepended to them or occur in a file in which the statement "`using namespace ord;`" occurs before the variable is referenced.

C++ code	Ordinal
`omega+12`	$\omega + 12$
`omega*3`	$\omega 3$
`omega*3 + 12`	$\omega 3 + 12$
`omega^5`	ω^5
`omega^(omega^12)`	$\omega^{\omega^{12}}$
`(omega^(omega^12)*6) + (omega^omega)*8 +12`	$\omega^{\omega^{12}6} + \omega^{\omega}8 + 12$

Table 1: `Ordinal` C++ code examples

`Ordinals` `two` through `six` are defined as members of `class Ordinal`[20].

The standard operations for ordinal arithmetic (`+`, `*` for $\times$ and `^` for exponentiation) are defined for all `Ordinal` instances. Expressions involving exponentiation must use parenthesis to indicate precedence because C++ gives lower precedence to `^` then it does to addition and multiplication[21]. In C++ the standard use of `^` is for the boolean operation exclusive or. Some examples of infinite `Ordinals` created by `Ordinal` expressions are shown in Table 1.

`Ordinals` that are a sum of terms are made up of a sequence of instances of the `class CantorNormalElement` each instance of this class contains an integer `factor` that multiplies the term and an `Ordinal` exponent of ω. For finite ordinals this exponent is 0.

4.1 `Ordinal::normalForm` and `Ordinal::texNormalForm`

Two `Ordinal` member functions, `normalForm` and `texNormalForm`, return a C++ `string` that can be output to display the value of an `Ordinal` in Cantor normal form or a variation of it defined here for ordinals $> \varepsilon_0$. `normalForm` creates a plain text format that is used for input and plain text output in the interactive mode of this program described in Appendix A. `texNormalForm` outputs a similar `string` in TEX math mode format. This `string` does *not* include the '`$`' markers to enter and exit TEX math mode. These must be added when this output is included in a TEX document. There are examples of this output in Section A.11.3 on **Display options**. Many of the entries in the tables in this manual are generated using `texNormalForm`.

4.2 `Ordinal::compare` member function

The compare function has a single `Ordinal` as an argument. It compares the `Ordinal` instance it is a member of with its argument. It scans the terms of both `Ordinals` (see equation 1) in order of decreasing significance. The `exponent`, α_k and then the `factor` n_k are compared. If these are both equal the comparison proceeds to the next term of both ordinals. `compare` is called recursively to compare the exponents (which are `Ordinals`) until it resolves to comparing an integer with an infinite ordinal or another integer. It returns 1, 0 or -1 if the ordinal called from is greater than, equal to or less than its argument.

[20]Reference to members of `class Ordinal` must include the prefix "`Ordinal::`" except in member functions of `Ordinal`

[21]The interactive mode of entering ordinal expressions (see Appendix A) has the desired precedence and does not require parenthesis to perform exponentiation before multiplication.

$\alpha = \sum_{m=0,1,\ldots,k} \omega^{\alpha_m} n_m$ from equation 1	
$\gamma = \sum_{m=0,1,\ldots,k-1} \omega^{\alpha_m} n_m + \omega^{\alpha_k}(n_k - 1)$	
Last Term ($\omega^{\alpha_k} n_k$) Condition	α.`limitElement(i)`
$\alpha_k = 0$ (α is not a limit)	UNDEFINED ABORT
$\alpha_k = 1$	$\gamma + i$
$\alpha_k > 1 \land \alpha_k$ is a successor	$\gamma + \omega^{\alpha_k - 1} i$
α_k is a limit	$\gamma + \omega^{(\alpha_k).\texttt{limitElement(i)}}$

Table 2: Cases for computing `Ordinal::limitElement`

`Ordinal`	`limitElement`				
ω	1	2	10	100	786
$\omega 8$	$\omega 7 + 1$	$\omega 7 + 2$	$\omega 7 + 10$	$\omega 7 + 100$	$\omega 7 + 786$
ω^2	ω	$\omega 2$	$\omega 10$	$\omega 100$	$\omega 786$
ω^3	ω^2	$\omega^2 2$	$\omega^2 10$	$\omega^2 100$	$\omega^2 786$
ω^ω	ω	ω^2	ω^{10}	ω^{100}	ω^{786}
$\omega^{\omega+2}$	$\omega^{\omega+1}$	$\omega^{\omega+1} 2$	$\omega^{\omega+1} 10$	$\omega^{\omega+1} 100$	$\omega^{\omega+1} 786$
$\omega^{\omega^{\omega^\omega}}$	ω^{ω^ω}	$\omega^{\omega^{\omega^2}}$	$\omega^{\omega^{\omega^{10}}}$	$\omega^{\omega^{\omega^{100}}}$	$\omega^{\omega^{\omega^{786}}}$

Table 3: `Ordinal::limitElement` examples.

Each term of an ordinal (from Equation 1) is represented by an instance of class `CantorNormalElement` and the bulk of the work of `compare` is done in member function `CantorNormalElement::compare` This function compares two terms of the Cantor normal form of an ordinal.

4.3 `Ordinal::limitElement` member function

`Ordinal` member function `limitElement` has a single integer parameter. It is only defined for notations of limit ordinals and will abort if called from a successor `Ordinal`. Larger values of this argument produce larger ordinal notations as output. The union of the ordinals represented by the outputs for all integer inputs is equal to the ordinal represented by the `Ordinal` instance `limitElement` is called from. This function satisfies requirement 3 on page 6.

In the following description mathematical notation is mixed with C++ code. Thus `limitElement(i)` called from an `Ordinal class` instance that represents ω^α is written as (ω^α).`limitElement(i)`.

The algorithm for `limitElement` uses the Cantor normal form in equation 1. The kernel processing is done in `CantorNormalElement::limitElement`. `limitElement` operates on the last or least significant term of the normal form. γ is used to represent all terms but the least significant. If the lest significant term is infinite and has a factor, the n_k in equation 1, greater than 1, γ will also include the term $\omega^{\alpha_k}(n_k - 1)$. The outputs from `limitElement` are γ and a final term that varies according to the conditions in Table 2. Table 3 gives some examples.

Operation	Example	Description
addition	$\alpha + \beta$	add 1 to α β times
multiplication	$\alpha \times \beta$	add α β times
exponentiation	α^{β}	multiply α β times
nested exponentiation	$\alpha^{\beta^{\gamma}}$	multiply α β^{γ} times
...	...	...

Table 4: Base `class Ordinal` operators.

Expression	Cantor Normal Form	C++ code
$(\omega 4 + 12)\omega$	ω^2	`(omega*4+12)*omega`
$\omega(\omega 4 + 12)$	$\omega^2 4 + \omega 12$	`omega*(omega*4+12)`
$\omega^{\omega(\omega+3)}$	$\omega^{\omega^2 + \omega 3}$	`omega∧(omega*(omega+3))`
$(\omega + 4)(\omega + 5)$	$\omega^2 + \omega 5 + 4$	`(omega+4)*(omega+5)`
$(\omega + 2)^{(\omega+2)}$	$\omega^{\omega+2} + \omega^{\omega+1} 2 + \omega^{\omega} 2$	`(omega+2)∧(omega+2)`
$(\omega + 3)^{(\omega+3)}$	$\omega^{\omega+3} + \omega^{\omega+2} 3 +$ $\omega^{\omega+1} 3 + \omega^{\omega} 3$	`(omega+3)∧(omega+3)`

Table 5: Ordinal arithmetic examples

4.4 `Ordinal` operators

The operators in the base class are built on the successor (or '+1') operation and recursive iteration of that operation. These are shown in Table 4.

The operators are addition, multiplication and exponentiation. These are implemented as C++ overloaded operators: `+`, `*` and `^`[22] Arithmetic on the ordinals is not commutative, $3 + \omega = \omega \neq \omega + 3$, For this and other reasons, caution is required in writing expressions in C++. Precedence and other rules used by the compiler are incorrect for ordinal arithmetic. It is safest to use parenthesis to completely specify the intended operation. Some examples are shown in Table 5.

4.4.1 `Ordinal` addition

In ordinal addition, all terms of the first operand that are at least a factor of ω smaller than the leading term of the second cam be ignored because of the following:

$$\alpha, \beta \ ordinals \ \wedge \alpha \geq \beta \rightarrow \beta + \alpha * \omega = \alpha * \omega \qquad (2)$$

`Ordinal` addition operates in sequence on the terms of both operands. It starts with the most significant terms. If they have the same exponents. their factors are added. Otherwise, if the second operand's exponent is larger than the first operand's exponent, the remainder of the first operand is ignored. Alternatively, if the second operands most significant exponent is less than the first operands most significant exponent, the leading term of the first operand

[22]'^' is used for 'exclusive or' in C++ and has *lower* precedence than any arithmetic operator such as '+'. Thus C++ will evaluate `x^y+1` as `x^(y+1)`. Use parenthesis to override this as in `(x^y)+1`.

is added to the result. The remaining terms are compared in the way just described until all terms of both operands have been dealt with.

4.4.2 `Ordinal` multiplication

Multiplication of infinite ordinals is complicated by the way addition works. For example:

$$(\omega + 3) \times 2 = (\omega + 3) + (\omega + 3) = \omega \times 2 + 3 \tag{3}$$

Like addition, multiplication works with the leading (most significant) terms of each operand in sequence. The operation that takes the product of terms is a member function of base class `CantorNormalElement`. It can be overridden by subclasses without affecting the algorithm than scans the terms of the two operands. When a subclass of `Ordinal` is added a subclass of `CantorNormalElement` must also be added.

`CantorNormalElement` and each of its subclasses is assigned a `codeLevel` that grows with the depth of `class` nesting. `codeLevel` for a `CantorNormalElement` is `cantorCodeLevel`. Any Cantor normal form term that is of the form ω^α will be at this level regardless of the level of the terms of α. `codeLevel` determines when a higher level function needs to be invoked. For example if we multiply α at `cantorCodeLevel` by β at a higher level then a higher level routine must be used. This is accomplished by calling β`.multiplyBy(`α`)` which will invoke the `virtual` function `multiplyBy` in the subclass β is an instance of.

The routine that multiplies two terms or `CantorNormalElement`s first tests the `codeLevel` of its operand and calls `multiplyBy` if necessary. If both operands are at `cantorCodeLevel`, the routine checks if both operands are finite and, if so, returns their integer product. If the first operand is finite and the second is infinite, the second operand is returned unchanged. All remaining cases are handled by adding the exponents of the two operands and multiplying their factors. The exponents are the α_i and the factors are the n_i in Equation 1. A `CantorNormalElement` with the computed exponent and factor is returned. If the exponents contain terms higher then `cantorCodeLevel`, this will be dealt with by the routine that does the addition of exponents.

The routine that multiplies single terms is called by a top level routine that scans the terms of the operands. If the second operand does not have a finite term, then only the most significant term of the first operand will affect the result by Equation 2. If the second operand does end in a finite term then all but the most significant term of the first operand, as illustrated by Equation 3, will be added to the result of multiplying the most significant term of the first operand by all terms of the second operand in succession. Some examples are shown in Table 6.

4.4.3 `Ordinal` exponentiation

`Ordinal` exponentiation first handles the cases when either argument is zero or one. It then checks if both arguments are finite and, if so, does an integer exponentiation[23]. If the base is finite and the exponent is infinite, the product of the infinite terms in the exponent is

[23]A large integer exponent can require more memory than is available to store the result and abort the program.

α	β	$\alpha \times \beta$
$\omega+1$	$\omega+1$	$\omega^2+\omega+1$
$\omega+1$	$\omega+2$	$\omega^2+\omega 2+1$
$\omega+1$	ω^3	ω^4
$\omega+1$	$\omega^3 2+2$	$\omega^4 2+\omega 2+1$
$\omega+1$	$\omega^4+\omega^3+\omega 7+3$	$\omega^5+\omega^4+\omega^2 7+\omega 3+1$
$\omega+1$	$\omega^\omega 3$	$\omega^\omega 3$
$\omega+2$	$\omega+1$	$\omega^2+\omega+2$
$\omega+2$	$\omega+2$	$\omega^2+\omega 2+2$
$\omega+2$	ω^3	ω^4
$\omega+2$	$\omega^3 2+2$	$\omega^4 2+\omega 2+2$
$\omega+2$	$\omega^4+\omega^3+\omega 7+3$	$\omega^5+\omega^4+\omega^2 7+\omega 3+2$
$\omega+2$	$\omega^\omega 3$	$\omega^\omega 3$
ω^3	$\omega+1$	$\omega^4+\omega^3$
ω^3	$\omega+2$	$\omega^4+\omega^3 2$
ω^3	ω^3	ω^6
ω^3	$\omega^3 2+2$	$\omega^6 2+\omega^3 2$
ω^3	$\omega^4+\omega^3+\omega 7+3$	$\omega^7+\omega^6+\omega^4 7+\omega^3 3$
ω^3	$\omega^\omega 3$	$\omega^\omega 3$
$\omega^3 2+2$	$\omega+1$	$\omega^4+\omega^3 2+2$
$\omega^3 2+2$	$\omega+2$	$\omega^4+\omega^3 4+2$
$\omega^3 2+2$	ω^3	ω^6
$\omega^3 2+2$	$\omega^3 2+2$	$\omega^6 2+\omega^3 4+2$
$\omega^3 2+2$	$\omega^4+\omega^3+\omega 7+3$	$\omega^7+\omega^6+\omega^4 7+\omega^3 6+2$
$\omega^3 2+2$	$\omega^\omega 3$	$\omega^\omega 3$
$\omega^4+\omega^3+\omega 7+3$	$\omega+1$	$\omega^5+\omega^4+\omega^3+\omega 7+3$
$\omega^4+\omega^3+\omega 7+3$	$\omega+2$	$\omega^5+\omega^4 2+\omega^3+\omega 7+3$
$\omega^4+\omega^3+\omega 7+3$	ω^3	ω^7
$\omega^4+\omega^3+\omega 7+3$	$\omega^3 2+2$	$\omega^7 2+\omega^4 2+\omega^3+\omega 7+3$
$\omega^4+\omega^3+\omega 7+3$	$\omega^4+\omega^3+\omega 7+3$	$\omega^8+\omega^7+\omega^5 7+\omega^4 3+\omega^3+\omega 7+3$
$\omega^4+\omega^3+\omega 7+3$	$\omega^\omega 3$	$\omega^\omega 3$
$\omega^\omega 3$	$\omega+1$	$\omega^{\omega+1}+\omega^\omega 3$
$\omega^\omega 3$	$\omega+2$	$\omega^{\omega+1}+\omega^\omega 6$
$\omega^\omega 3$	ω^3	$\omega^{\omega+3}$
$\omega^\omega 3$	$\omega^3 2+2$	$\omega^{\omega+3}2+\omega^\omega 6$
$\omega^\omega 3$	$\omega^4+\omega^3+\omega 7+3$	$\omega^{\omega+4}+\omega^{\omega+3}+\omega^{\omega+1}7+\omega^\omega 9$
$\omega^\omega 3$	$\omega^\omega 3$	$\omega^{\omega 2}3$

Table 6: `Ordinal` multiply examples

Expression	Cantor Normal Form	C++ code
4^{ω^7+3}	$\omega^7 64$	`Ordinal four(4);` `four^((omega^7)+3)`
$5^{\omega^7+\omega+3}$	$\omega^8 125$	`five^` `((omega^7)+omega+3)`
$(\omega+1)^4$	$\omega^4+\omega^3+\omega^2+\omega+1$	`(omega+1)^4`
$(\omega^2+\omega+3)^{\omega^2+\omega+1}$	$\omega^{\omega^2+\omega+2}+\omega^{\omega^2+\omega+1}+$ $\omega^{\omega^2+\omega}3$	`((omega^2)+omega+3)^` `((omega^2)+omega+1)`
$(\omega^2+\omega+3)^{\omega^2+\omega+2}$	$\omega^{\omega^2+\omega+4}+\omega^{\omega^2+\omega+3}+$ $\omega^{\omega^2+\omega+2}3+\omega^{\omega^2+\omega+1}+\omega^{\omega^2+\omega}3$	`((omega^2)+omega+3)^` `((omega^2)+omega+2)`

Table 7: C++ `Ordinal` exponentiation examples

computed. If the exponent has a finite term, this product is multiplied by the base taken to the power of this finite term. This product is the result.

If the base is infinite and the exponent is an integer, n, the base is multiplied by itself n times. To do this efficiently, all powers of two less than n are computed. The product of those powers of 2 necessary to generate the result is computed.

If both the base and exponent are infinite, then the infinite terms of the exponent are scanned in decreasing sequence. Each is is used as an exponent applied to the most significant term of the base. The sequence of exponentials is multiplied. If the exponent has a finite term then the entire base, not just the leading term, is raised to this finite power using the algorithm described above for a finite exponent and infinite base. That factor is then applied to the previous product of powers.

To compute the above result requires a routine for taking the exponential of a single infinite term of the Cantor normal form i. e. a `CantorNormalElement` (see Equation 1) by another infinite single term. (When `Ordinal` subclasses are defined this is the only routine that must be overridden.) The algorithm is to multiply the exponent of ω from the first operand (the base) by the second operand, That product is used as the exponent of ω in the term returned. Table 7 gives some examples with C++ code and some additional examples are shown in Table 8.

5 The Veblen hierarchy

This section gives a brief overview of the Veblen hierarchy and the Δ operator. See [15, 11, 6] for a more a complete treatment. This is followed by the development of a computational approach for constructing notations for these ordinals up to (but not including) the large Veblen ordinal. We go further in Sections 8 and 9.

The Veblen hierarchy extends the recursive ordinals beyond ε_0. A Veblen hierarchy can be constructed from any strictly increasing continuous function[24] f, whose domain and range

[24] A continuous function, f, on the ordinals must map limits to limits. Thus for every infinite limit ordinal y, $f(y) = sup\{f(v) : v < y\}$. A continuous strictly increasing function on the ordinals is called a normal function.

α	β	α^β
$\omega+1$	$\omega+1$	$\omega^{\omega+1}+\omega^\omega$
$\omega+1$	$\omega+2$	$\omega^{\omega+2}+\omega^{\omega+1}+\omega^\omega$
$\omega+1$	ω^3	ω^{ω^3}
$\omega+1$	$\omega^3 2+2$	$\omega^{\omega^3 2+2}+\omega^{\omega^3 2+1}+\omega^{\omega^3 2}$
$\omega+1$	ω^4+3	$\omega^{\omega^4+3}+\omega^{\omega^4+2}+\omega^{\omega^4+1}+\omega^{\omega^4}$
$\omega+1$	$\omega^\omega 3$	$\omega^{\omega^\omega 3}$
$\omega+2$	$\omega+1$	$\omega^{\omega+1}+\omega^\omega 2$
$\omega+2$	$\omega+2$	$\omega^{\omega+2}+\omega^{\omega+1}2+\omega^\omega 2$
$\omega+2$	ω^3	ω^{ω^3}
$\omega+2$	$\omega^3 2+2$	$\omega^{\omega^3 2+2}+\omega^{\omega^3 2+1}2+\omega^{\omega^3 2}2$
$\omega+2$	ω^4+3	$\omega^{\omega^4+3}+\omega^{\omega^4+2}2+\omega^{\omega^4+1}2+\omega^{\omega^4}2$
$\omega+2$	$\omega^\omega 3$	$\omega^{\omega^\omega 3}$
ω^3	$\omega+1$	$\omega^{\omega+3}$
ω^3	$\omega+2$	$\omega^{\omega+6}$
ω^3	ω^3	ω^{ω^3}
ω^3	$\omega^3 2+2$	$\omega^{\omega^3 2+6}$
ω^3	ω^4+3	ω^{ω^4+9}
ω^3	$\omega^\omega 3$	$\omega^{\omega^\omega 3}$
$\omega^3 2+2$	$\omega+1$	$\omega^{\omega+3}2+\omega^\omega 2$
$\omega^3 2+2$	$\omega+2$	$\omega^{\omega+6}2+\omega^{\omega+3}4+\omega^\omega 2$
$\omega^3 2+2$	ω^3	ω^{ω^3}
$\omega^3 2+2$	$\omega^3 2+2$	$\omega^{\omega^3 2+6}2+\omega^{\omega^3 2+3}4+\omega^{\omega^3 2}2$
$\omega^3 2+2$	ω^4+3	$\omega^{\omega^4+9}2+\omega^{\omega^4+6}4+\omega^{\omega^4+3}4+\omega^{\omega^4}2$
$\omega^3 2+2$	$\omega^\omega 3$	$\omega^{\omega^\omega 3}$
ω^4+3	$\omega+1$	$\omega^{\omega+4}+\omega^\omega 3$
ω^4+3	$\omega+2$	$\omega^{\omega+8}+\omega^{\omega+4}3+\omega^\omega 3$
ω^4+3	ω^3	ω^{ω^3}
ω^4+3	$\omega^3 2+2$	$\omega^{\omega^3 2+8}+\omega^{\omega^3 2+4}3+\omega^{\omega^3 2}3$
ω^4+3	ω^4+3	$\omega^{\omega^4+12}+\omega^{\omega^4+8}3+\omega^{\omega^4+4}3+\omega^{\omega^4}3$
ω^4+3	$\omega^\omega 3$	$\omega^{\omega^\omega 3}$
$\omega^\omega 3$	$\omega+1$	$\omega^{\omega^2+\omega}3$
$\omega^\omega 3$	$\omega+2$	$\omega^{\omega^2+\omega 2}3$
$\omega^\omega 3$	ω^3	ω^{ω^4}
$\omega^\omega 3$	$\omega^3 2+2$	$\omega^{\omega^4 2+\omega 2}3$
$\omega^\omega 3$	ω^4+3	$\omega^{\omega^5+\omega 3}3$
$\omega^\omega 3$	$\omega^\omega 3$	$\omega^{\omega^\omega 3}$

Table 8: `Ordinal` exponential examples

are the countable ordinals such that $f(0) > 0$. $f(x) = \omega^x$ satisfies these conditions and is the starting point for constructing the standard Veblen hierarchy. One core idea is to define a new function from an existing one so that the new function enumerates the fixed points of the first one. A fixed point of f is a value v such that $f(v) = v$. Given an infinite sequence of such functions one can define a new function that enumerates the *common* fixed points of all functions in the sequence. In this way one can iterate the construction of a new function up to any countable ordinal. The Veblen hierarchy based on $f(x) = \omega^x$ is written as $\varphi(\alpha, \beta)$ and defined as follows.

$\varphi(0, \beta) = \omega^\beta$.

$\varphi(\alpha + 1, \beta)$ enumerates the fixed points of $\varphi(\alpha, \beta)$.

$\varphi(\alpha, \beta)$ for α a limit ordinal, α, enumerates the intersection of the fixed points of $\varphi(\gamma, \beta)$ for γ less than α.

From a full Veblen hierarchy one can define a diagonalization function $\varphi(x, 0)$ from which a new Veblen hierarchy can be constructed. This can be iterated and the Δ operator does this in a powerful way.

5.1 The delta operator

The Δ operator is defined as follows[11, 6].

- $\Delta_0(\psi)$ enumerates the fixed points of the normal (continuous and strictly increasing) function on the ordinals ψ.
- $\Delta_{\alpha'}(\varphi) = \Delta_0(\varphi^\alpha(-, 0))$. That is it enumerates the fixed points of the diagonalization of the Veblen hierarchy constructed from φ^α.
- $\Delta_\alpha(\varphi)$ for α a limit ordinal enumerates $\bigcap_{\gamma<\alpha}$ range $\Delta_\gamma(\varphi)$.
- $\varphi_0^\alpha = \varphi$.
- $\varphi_{\beta'}^\alpha = \Delta_\alpha(\varphi_\beta^\alpha)$.
- φ_β^α for β a limit ordinal enumerates $\bigcap_{\gamma<\beta}$ range φ_γ^α.

The function that enumerates the fixed points of a base function is a function on functions. The Veblen hierarchy is constructed by iterating this function on functions starting with ω^x. A generalized Veblen hierarchy is constructed by a similar iteration starting with any function on the countable ordinals, $f(x)$, that is strictly increasing and continuous (see Note 24) with $f(0) > 0$. The Δ operator defines a higher level function. Starting with the function on functions used to define a general Veblen hierarchy, it defines a hierarchy of functions on functions. The Δ operator constructs a higher level function that builds and then diagonalizing a Veblen hierarchy.

In a computational approach, such functions can only be partially defined on objects in an always expandable computational framework. The `class`es in which the functions are defined and the functions themselves are designed to be extensible as future subclasses are added to the system.

5.2 A finite function hierarchy

An obvious extension called the Veblen function is to iterate the functional hierarchy any finite number of times. This can be represented as a function on ordinal notations,

$$\varphi(\alpha_1, \alpha_2, ..., \alpha_n). \tag{4}$$

Each of the parameters is an ordinal notation and the function evaluates to an ordinal notation. The first parameter is the most significant. It represents the iteration of the highest level function. Each successive ordinal[25] operand specifies the level of iteration of the next lowest level function. With a single parameter Equation 4 is the function ω^x ($\varphi(\alpha) = \omega^\alpha$). With two parameters, $\varphi(\alpha, \beta)$, it is the Veblen hierarchy constructed from the base function ω^x. With three parameters we have the Δ hierarchy built on this initial Veblen hierarchy. In particular the following holds.

$$\varphi(\alpha, \beta, x) = \Delta_\alpha \varphi^\alpha_\beta(x) \tag{5}$$

5.3 The finite function normal form

The Veblen function and its extension to a function with an arbitrary finite number of parameters requires the following extension to the Cantor Normal Form in Equation 1.

$$\alpha = \alpha_1 n_1 + \alpha_2 n_2 + \alpha_3 n_3 + ... + \alpha_k n_k \tag{6}$$

$$\alpha_i = \varphi(\beta_{1,i}, \beta_{2,i}, ..., \beta_{m_i,i})$$

$$k \geq 1, k \geq i \geq 1, m_i \geq 1, n_k \geq 1, \alpha_1 > \alpha_2 > \alpha_3 > ... > \alpha_k$$

α_i and $\beta_{i,j}$ are ordinals; i, j, k, m_i, n_k are integers.

Note that $\varphi(\beta) = \omega^\beta$ so the above includes the Cantor normal form terms. To obtain a unique representation for each ordinal the rule is adopted that all normal forms must be reduced to the simplest expression that represents the same ordinal. For example $\varphi(1, 0) = \varepsilon_0 = \omega^{\varepsilon_0} = \varphi(\varphi(1, 0))$. This requires that fixed points be detected and reduced as described in 6.3.

In this computational approach, the meaning of ordinal notations is defined by functions `compare` and `limitElement`. What `compare` does is determined by `limitElement` which defines each notation in terms of notations for smaller ordinals.

[25] All references to ordinals in the context of describing the computational approach refer to ordinal notations. The word notation will sometimes be omitted when it is obviously meant and would be tedious to keep repeating.

$\alpha = \varphi(\beta_1, \beta_2, ..., \beta_m)$ **from Equation 6.**			
`lp1`**,** `rep1` **and** `rep2` **abbreviate** `limPlus_1` `replace1` **and** `replace2`**.**			
Conditions on the least significant non zero parameters, `leastOrd` (index `least`) and `nxtOrd` (index `nxt`)			Routines `rep1` and `rep2` replace 1 or 2 parameters in Equation 6. The index and value to replace (one or two instances) are the parameters to these routines. `lp1()` adds one to an ordinal with `psuedoCodeLevel` > `cantorCodeLevel` (see Section 6.3 on fixed points). `ret` is the result returned.
X	`nxtOrd`	`leastOrd`	`ret=`α`.limitElement(i)`
A	ignore	limit[1]	`ret=rep1(least,leastOrd.limitElement(i).lp1());`
B	ignore	successor[2]	`ret=rep2(least,leastOrd-1,least+1,1);` `for (int j=1; j<i; j++)` `ret= rep2(least,leastOrd-1,least+1,ret.lp1());`
C	limit	successor[3]	`tmp=rep1(least,leastOrd-1).lp1();` `ret=rep2(nxt,nxtOrd.limitElement(i).lp1(),` `nxt+1,tmp);`
D	successor	successor[3]	`ret=rep1(least,leastOrd-1).lp1();` `for (int j=1; j<i; j++)` `ret=rep2(nxt, nxtOrd-1, nxt+1,ret.lp1());`

The '**X**' column gives the exit code from `limitElement` (see Section 6.2).
1 least significant non zero parameter may or may not be least significant.
2 least significant non zero parameter is not least significant.
3 least significant non zero parameter is least significant.

Table 9: Cases for computing $\varphi(\beta_1, \beta_2, ..., \beta_m)$.`limitElement(i)`

5.4 `limitElement` for finite functions

The `LimitElement` member function for an ordinal notation `ord` defines `ord` by enumerating notations for smaller ordinals such that the union of the ordinals those notations represent is the ordinal represented by `ord`. As with base `class Ordinal` all but the least significant term is copied unchanged to each output of `limitElement`. Table 2 for `Ordinal::limitElement` specifies how terms, excluding the least significant and the factors (the n_i in Equation 1), are handled. This does not change for the extended normal form in Equation 6. Table 9 extends Table 2 by specifying how the least significant normal form term (if it is $\geq \varepsilon_0$) is handled in constructing `limitElement(i)`. If the factor of this term is greater than 1 or there are other terms in the ordinal notation then the algorithms from Table 2 must also be used in computing the final output. The idea in defining how `limitElement` works on the least significant term is to construct an infinite sequence that is the the strongest possible iteration of allowed processes and notations.

Table 9 uses pseudo C++ code adapted from the implementation of `limitElement`. Variable names have been shortened to limit the size of the table and other simplifications have been made. However the code accurately describes the logic of the program. Variable `ret` is the result or output of the subroutine. Different sequences are generated based on the two least significant non zero parameters of φ in Equation 4 and whether the least significant non zero term is the least significant term (including those that are zero). The idea is to construct an infinite sequence with a limit that is not reachable with a finite sequence of smaller notations.

5.5 An iterative functional hierarchy

A finite functional hierarchy with an arbitrarily large number of parameters can be expanded with a limit that is a sequence of finite functionals with an ever increasing number of parameters. Using this as the successor operation and taking the union of all hierarchies defined by a limit ordinal allows iteration of a functional hierarchy up to any recursive ordinal. The key to defining this iteration is the `limitElement` member function.

To support this expanded notation the normal form in Equation 6 is expanded as follows.

$$\alpha = \alpha_1 n_1 + \alpha_2 n_2 + \alpha_3 n_3 + ... + \alpha_k n_k$$

$$\alpha_i = \varphi_{\gamma_i}(\beta_{1,i}, \beta_{2,i}, ..., \beta_{m_i,i}) \tag{7}$$

$$k \geq 1, k \geq i \geq 1, m_i \geq 1, n_k \geq 1, \alpha_1 > \alpha_2 > \alpha_3 > ... > \alpha_k$$

$$\alpha_i \text{ and } \beta_{i,j} \text{ are ordinals; } i, j, k, m_i, n_k \text{ are integers.}$$

γ_i, the subscript to φ, is the ordinal the functional hierarchy is iterated up to. $\varphi_0(\beta_1, \beta_2, ..., \beta_m)$ $= \varphi(\beta_1, \beta_2, ..., \beta_n)$. $\varphi_1(0) = \varphi_1$ is the notation for an infinite union of the ordinals represented by finite functionals. Specifically it represents the union of ordinals with notations: $\varphi(1), \varphi(1,0).\varphi(1,0,0), ...,$. $\varphi(1) = \omega, \varphi(1,0) = \varepsilon_0$ and $\varphi(1,0,0) = \Gamma_0$. $\varphi_0(\alpha) = \varphi(\alpha) = \omega^\alpha$ and $\varphi_{\gamma+1} = \varphi_\gamma(1) \cup \varphi_\gamma(1,0) \cup \varphi_\gamma(1,0,0), ...,$.

The definition of `limitElement` for this hierarchy is shown in Table 10. This is an extension of Table 9. That table and the definition of `compare` (See Section 7.1) define the notations represented by Equation 7. The sub`class FiniteFuncOrdinal` (Section 6) defines finite functional notations for recursive ordinals. The sub`class IterFuncOrdinal` (Section 7) defines iterative functional notations for recursive ordinals.

6 `FiniteFuncOrdinal class`

`FiniteFuncOrdinal class` is derived from `Ordinal` base class. It implements ordinal notations for the normal form in Equation 6. Each term or $\alpha_i n_i$ is defined by an instance of `class FiniteFuncNormalElement` or `CantorNormalElement`. Any term that is a `FiniteFuncNormalElement` will have `codeLevel` set to `finiteFuncCodeLevel`.

The `FiniteFuncOrdinal class` should not be used directly to create ordinal notations. Instead use functions `psi` or `finiteFunctional`[26]. `psi` constructs notations for the initial Veblen hierarchy. It requires exactly two parameters (for the single parameter case use $\varphi(\alpha) = \omega^\alpha$). `finiteFunctional` accepts 3 to 5 parameters. For more than 5 use a

[26]In the interactive ordinal calculator the `psi` function can be use with any number of parameters to define a `FiniteFuncOrdinal`. See Section A.11.5 for some examples.

$\alpha = \varphi_\gamma(\beta_1, \beta_2, ..., \beta_m)$ from Equation 7. For this table $m = 1$.		
X	Condition	α.`limitElement(i)`
E	$\beta_1 = 0, \gamma$ limit	$\varphi_{\gamma\texttt{limitElement(i).lp1()}}(0)$
F	$\beta_1 = 0, \gamma$ successor	$\varphi_{\gamma-1}(\varphi_{\gamma-1}(0), 0, 0, ..., 0)$ (i parameters)
G	β_1 limit	$\varphi_\gamma(\beta_1.\texttt{limitElement(i).lp1()})$
H	β_1 successor γ limit	$\varphi_{\gamma.\texttt{limitElement(i).lp1()}}(\varphi_\gamma(\beta_1 - 1).\texttt{lp1}(), 0, 0, ..., 0)$ (i parameters)
I	β_1 successor γ successor	$\varphi_{\gamma-1}(\varphi_\gamma(\beta_1 - 1).\texttt{lp1}(), 0, 0, ..., 0)$ (i parameters)
J	$m > 1$	See Table 9 for more than one β_i parameter.

The '**X** column gives the exit code from `limitElement` (see Section 6.2).
If $\beta_1 = 0$, it is the only β_i. Leading zeros are normalized away.

Table 10: Cases for computing $\varphi_\gamma(\beta_1, \beta_2, ..., \beta_m)$.`limitElement(i)`

`NULL` terminated array of pointers to `Ordinals` or `createParameters` to create this array. `createParameters` can have 1 to 9 parameters all of which must be pointers to `Ordinals`.

Some examples are show in Table 11. The direct use of the `FiniteFuncOrdinal` constructor is shown in Table 12. The '`Ordinal`' column of both tables is created using `Ordinal::texNormalForm` which uses the standard notation for ε_α and Γ_α where appropriate. These functions reduce fixed points to their simplest expression and declare an `Ordinal` instead of a `FiniteFuncOrdinal` if appropriate. The first line of Table 11 is an example of this.

`FiniteFuncOrdinal` can be called with 3 or 4 parameters. For additional parameters, it can be called with a `NULL` terminated array of pointers to `Ordinal` notations. `createParameters` can be used to create this array as shown in the last line of Table 11[27].

6.1 `FiniteFuncNormalElement::compare` member function

Because of virtual functions, there is no need for `FiniteFuncOrdinal::compare`. The work of comparing the sequence of terms in Equation 6 is done by `Ordinal::compare` and routines it calls. `FiniteFuncNormalElement::compare` is automatically called for comparing individual terms in the normal form from Equation 6. It overrides `CantorNormalElement::compare`. It outputs 1, 0 or -1 if the object it is called from is greater than equal to or less than its argument term. The $\beta_{j,i}$ in Equation 6 are represented by the elements of array `funcParameters` in C++.

The `FiniteFuncNormalElement` object `compare` (or any `class` member function) is called from is '`this`' in C++. The `CantorNormalElement` argument to `compare` is `trm`. If `trm.codeLevel` > `finiteFuncCodeLevel`[28] then `-trm.compare(*this)` is returned. This invokes the subclass member function associated with the sub`class` and `codeLevel` of `trm`.

`CantorNormalElement::compare` only needs to test the `exponent`s and if those are equal the `factor`s of the two normal form terms being compared. An arbitrarily large number

[27]The '&' character in the last line in Table 11 is the C++ syntax that constructs a pointer to the object '&' precedes.

[28] `finiteFuncCodeLevel` is the `codeLevel`(see Section 4.4.2) of a `FiniteFuncNormalElement`.

C++ code	**Ordinal**
`psi(zero,omega)`	ω^ω
`psi(one,zero)`	ε_0
`psi(one,one)`	ε_1
`psi(eps0,one)`	$\varphi(\varepsilon_0, 1)$
`psi(one,Ordinal::two)`	ε_2
`finiteFunctional(one,zero,zero)`	Γ_0
`finiteFunctional(one,zero,one)`	$\varphi(1, 0, 1)$
`finiteFunctional(one,zero,Ordinal::two)`	$\varphi(1, 0, 2)$
`finiteFunctional(one,Ordinal::two,zero)`	Γ_2
`finiteFunctional(one,zero,omega)`	$\varphi(1, 0, \omega)$
`finiteFunctional(one,eps0,omega)`	$\varphi(1, \varepsilon_0, \omega)$
`finiteFunctional(one,zero,zero,zero)`	$\varphi(1, 0, 0, 0)$
`finiteFunctional(createParameters(` `&one,&zero,&zero,&zero,&zero))`	$\varphi(1, 0, 0, 0, 0)$

Table 11: `finiteFunctional` C++ code examples

C++ code	**Ordinal**
`const Ordinal * const params[] =` `{&Ordinal::one,&Ordinal::zero,0};` `const FiniteFuncOrdinal eps0(params);`	ε_0
`Ordinal eps0_alt = psi(1,0);`	ε_0
`const FiniteFuncOrdinal eps0_alt2(1,0);`	ε_0
`const FiniteFuncOrdinal gamma0(1,0,0);`	Γ_0
`const FiniteFuncOrdinal gammaOmega(omega,0,0);`	$\varphi(\omega, 0, 0)$
`const FiniteFuncOrdinal gammax(gammaOmega,gamma0,omega);`	$\varphi(\varphi(\omega, 0, 0), \Gamma_0, \omega)$
`const FiniteFuncOrdinal big(1,0.0,0);`	$\varphi(1, 0, 0, 0)$

Table 12: `FiniteFuncOrdinal` C++ code examples

of `Ordinal` parameters are used to construct a `FiniteFuncNormalElement`. Thus a series of tests is required. This is facilitated by a member function `CantorNormalElement::getMaxParameter` that returns the largest parameter used in constructing this normal form term[29]. If `trm.codeLevel` $<$ `finiteFuncCodeLevel` then `trm` $>$ `this` only if the maximum parameter of `trm` is greater than `this`. However the value of `factor` for `this` must be ignored in making this comparison because `trm` $\geq \omega^{\texttt{trm.getMaxParameter()}}$ and this will swamp the effect of any finite `factor`.

The following describes `FiniteFuncNormalElement::compare` with a single `CantorNormalElement` parameter `trm`.

1. If `trm.codeLevel` $<$ `finiteFuncCodeLevel` the first (and thus largest) term of the exponent of the argument is compared to `this`, ignoring the two `factors`. If the result is nonzero that result is returned. Otherwise -1 is returned.

2. If the first term of the maximum parameter of the ordinal notation `compare` is called from is, ignoring factors, $\geq$ `trm`, return 1.

3. If `this` $\leq$ the maximum parameter of `trm` return -1.

If the above is not decisive `FiniteFuncNormalElement::compareFiniteParams` is called to compare in sequence the number of parameters and then the size of each parameter in succession starting with the most significant. If any difference is encountered that is returned as the result otherwise the result depends on the relative size of the two factors.

6.2 `FiniteFuncNormalElement::limitElement` member function

As with `compare` there is no need for `FiniteFuncOrdinal::limitElement`. The `Ordinal` member function is adequate. `FiniteFuncNormalElement::limitElement` overrides `CantorNormalElement::limitElement` described in Section 4.3. Thus it takes a single integer parameter. Increasing values for this argument yield larger ordinal notations as output. The union of the ordinals represented by the outputs for all integer inputs is equal to the ordinal represented by the `FiniteFuncNormalElement class` instance `limitElement` is called from. This will be referred to as the input term to `limitElement`.

`Ordinal::limitElement` copies all but the last term of the normal form of its input to the output it generates. For both `Ordinals` and `FiniteFuncOrdinals` this is actually done in `OrdinalImpl::limitElement`[30] The last term of the result is determined by a number of conditions on the last term of the input in `FiniteFuncNormalElement::limitElement`.

Tables 2 and 9 fully define `FiniteFuncOrdinal::limitElement`. The '**X**' column in Table 9 connect each table entry to the section of code preceding `RETURN1`. This is a debugging macro which has a quoted letter as a parameter. This letter is an exit code that matches the **X** column in Table 9. The C++ pseudo code in the table uses shorter variable names and takes other shortcuts, but accurately reflects the logic in the source code.

Some examples are shown in Table 13.

[29]For efficiency the constructor of a `FiniteFuncNormalElement` finds and saves the maximum parameter. For a `CantorNormalElement` the maximum parameter is the `exponent` as this is the only parameter that can be infinite. The case when the `factor` is larger than the `exponent` can be safely ignored.

[30]`OrdinalImpl` is an internal implementation `class` that does most of the work for instances of an `Ordinal`.

`FiniteFuncOrdinal`	`limitElement`		
ω	1	2	4
ε_0	ω	ω^ω	$\omega^{\omega^{\omega^\omega}}$
$\varphi(2,0)$	ε_1	$\varepsilon_{\varepsilon_1+1}$	$\varepsilon_{\varepsilon_{\varepsilon_{\varepsilon_1+1}+1}+1}$
$\varphi(\omega,\omega)$	$\varphi(\omega,1)$	$\varphi(\omega,2)$	$\varphi(\omega,4)$
Γ_0	ε_0	$\varphi(\varepsilon_0+1,0)$	$\varphi(\varphi(\varphi(\varepsilon_0+1,0)+1,0)+1,0)$
Γ_1	$\varphi(1,0,1)$	$\varphi(1,0,\varphi(1,0,1)+1)$	$\varphi(1,0,\varphi(1,0,\varphi(1,0,\varphi(1,0,1)+1)+1)+1)$
$\varphi(1,1,1)$	Γ_1+1	$\varphi(1,0,\Gamma_1+1)$	$\varphi(1,0,\varphi(1,0,\varphi(1,0,\Gamma_1+1)+1)+1)$
$\varphi(2,0,0)$	Γ_1	Γ_{Γ_1+1}	$\Gamma_{\Gamma_{\Gamma_{\Gamma_1+1}+1}+1}$
$\varphi(3,0,0)$	$\varphi(2,1,0)$	$\varphi(2,\varphi(2,1,0)+1,0)$	$\varphi(2,\varphi(2,\varphi(2,\varphi(2,1,0)+1,0)+1,0)+1,0)$
$\varphi(\omega,0)$	ε_0	$\varphi(2,0)$	$\varphi(4,0)$
$\varphi(\omega,1)$	$\varepsilon_{\varphi(\omega,0)+1}$	$\varphi(2,\varphi(\omega,0)+1)$	$\varphi(4,\varphi(\omega,0)+1)$
$\varphi(\omega,0,0)$	Γ_0	$\varphi(2,0,0)$	$\varphi(4,0,0)$
$\varphi(\omega,0,1)$	$\varphi(1,\varphi(\omega,0,0)+1,1)$	$\varphi(2,\varphi(\omega,0,0)+1,1)$	$\varphi(4,\varphi(\omega,0,0)+1,1)$
$\varphi(\omega,1,0)$	$\varphi(\omega,0,1)$	$\varphi(\omega,0,\varphi(\omega,0,1)+1)$	$\varphi(\omega,0,\varphi(\omega,0,\varphi(\omega,0,\varphi(\omega,0,1)+1)+1)+1)$
$\varphi(\omega,1,1)$	$\varphi(\omega,1,0)+1$	$\varphi(\omega,0,\varphi(\omega,1,0)+1)$	$\varphi(\omega,0,\varphi(\omega,0,\varphi(\omega,0,\varphi(\omega,1,0)+1)+1)+1)$
$\varphi(1,0,0,0)$	Γ_0	$\varphi(\Gamma_0+1,0,0)$	$\varphi(\varphi(\varphi(\Gamma_0+1,0,0)+1,0,0)+1,0,0)$
$\varphi(2,0,0,0)$	$\varphi(1,1,0,0)$	$\varphi(1,\varphi(1,1,0,0)+1,0,0)$	$\varphi(1,\varphi(1,\varphi(1,\varphi(1,1,0,0)+1,0,0)+1,0,0)+1,0,0)$
$\varphi(2,0,2,1)$	$\varphi(2,0,2,0)+1$	$\varphi(2,0,1,\varphi(2,0,2,0)+1)$	$\varphi(2,0,1,\varphi(2,0,1,\varphi(2,0,1,\varphi(2,0,2,0)+1)+1)+1)$
$\varphi(\omega,0,0,0)$	$\varphi(1,0,0,0)$	$\varphi(2,0,0,0)$	$\varphi(4,0,0,0)$
$\varphi(2,\varphi(2,0),\varepsilon_0)$	$\varphi(2,\varphi(2,0),\omega)$	$\varphi(2,\varphi(2,0),\omega^\omega)$	$\varphi(2,\varphi(2,0),\omega^{\omega^{\omega^\omega}})$
$\varphi(3,\varphi(2,0),\varepsilon_0)$	$\varphi(3,\varphi(2,0),\omega)$	$\varphi(3,\varphi(2,0),\omega^\omega)$	$\varphi(3,\varphi(2,0),\omega^{\omega^{\omega^\omega}})$
$\varphi(3,\varphi(2,0,0,0),\omega)$	$\varphi(3,\varphi(2,0,0,0),1)$	$\varphi(3,\varphi(2,0,0,0),2)$	$\varphi(3,\varphi(2,0,0,0),4)$
$\varphi(3,\varphi(2,0,0,0),\varepsilon_0)$	$\varphi(3,\varphi(2,0,0,0),\omega)$	$\varphi(3,\varphi(2,0,0,0),\omega^\omega)$	$\varphi(3,\varphi(2,0,0,0),\omega^{\omega^{\omega^\omega}})$
$\varphi(3,\varphi(2,0,0,0),\varepsilon_1)$	$\varphi(3,\varphi(2,0,0,0),\varepsilon_0+1)$	$\varphi(3,\varphi(2,0,0,0),\omega^{\varepsilon_0+1})$	$\varphi(3,\varphi(2,0,0,0),\omega^{\omega^{\omega^{\varepsilon_0+1}}})$
$\varphi(\varphi(2,0),\Gamma_0,\varepsilon_0)$	$\varphi(\varphi(2,0),\Gamma_0,\omega)$	$\varphi(\varphi(2,0),\Gamma_0,\omega^\omega)$	$\varphi(\varphi(2,0),\Gamma_0,\omega^{\omega^{\omega^\omega}})$

Table 13: `FiniteFuncOrdinal::limitElement` examples.

6.3 `FiniteFuncOrdinal::fixedPoint` member function

`FiniteFuncOrdinal::fixedPoint` is used by `finiteFunctional` to create an instance of `FiniteFuncOrdinal` in a normal form (Equation 6) that is the simplest expression for the ordinal represented. The routine has an index and an array of pointers to `Ordinal` notations as input. This array of notations contains the parameters for the notation being constructed. This function determines if the parameter at the specified index is a fixed point for a `FiniteFuncOrdinal` created with these parameters. If it is, `true` is returned and otherwise `false`. The routine that calls this routine selects the largest parameter in the array of `Ordinal` pointers as the one to check (the value of the index parameter). It then checks to see if all less significant parameters are 0. If not this cannot be a fixed point. `fixedPoint` is only called if this condition is met.

Section 4.4.1 defines the `codeLevel` assigned to a `CantorNormalElement`. From this a `psuedoCodeLevel` for an `Ordinal` is obtained by calling `Ordinal` member function with that name. `psuedoCodeLevel` returns `cantorCodeLevel` unless the ordinal notation normal form has a single term or `CantorNormalElement` with a factor of 1. In that case the `codeLevel` of that term is returned. This is helpful in evaluating fixed points because a parameter with a `psuedoCodeLevel` at `cantorCodeLevel` cannot be a fixed point.

If the parameter selected has `psuedoCodeLevel` $\leq$ `cantorCodeLevel`, `false` is returned. If the maximum parameter `psuedoCodeLevel` $>$ `finiteFuncCodeLevel`, `true` is returned. Otherwise a `FiniteFuncOrdinal` is constructed from all the parameters except that selected by the index which is set to zero[31]. If this value is less than the maximum parameter, `true` is returned and otherwise `false`.

6.4 `FiniteFuncOrdinal` operators

`FiniteFuncOrdinal` operators are extensions of the `Ordinal` operators defined in Section 4.4. No new code is required for addition.

6.4.1 `FiniteFuncOrdinal` multiplication

The code that combines the terms of a product for class `Ordinal` can be used without change for `FiniteFuncOrdinal`. The only routines that need to be overridden are those that take the product of two terms, i. e. two `CantorNormalElements` with at least one of these also being a `FiniteFuncNormalElement`. The two routines overridden are `multiply` and `multiplyBy`. Overriding these insures that, if *either* operand is a `FiniteFuncNormalElement` subclass of `CantorNormalElement`, the higher level `virtual` function will be called.

The key to multiplying two terms of the normal form representation, at least one of which is at `finiteFuncCodeLevel`, is the observation that every normal form term at `finiteFuncCodeLevel` with a `factor` of 1 is a fixed point of ω^x, i. e. $a = \omega^a$. Thus the product of two such terms, a and b is ω^{a+b}. Further the product of term a at this level and $b = \omega^\beta$ for any term b at `cantorCodeLevel` is $\omega^{a+\beta}$. Note in all cases if the first term

[31]If there is only one nonzero parameter (which must be the most significant), then the parameter array is increased by 1 and the most significant parameter is set to one in the value to compare with the maximum parameter.

has a `factor` other than 1 it will be ignored. The second term's `factor` will be applied to the result.

Multiply is mostly implemented in `FiniteFuncNormalElement::doMultiply` which is a `static` function[32] that takes both multiply arguments as operands. This routine is called by both `multiply` and `multiplyBy`. It first checks to insure that neither argument exceeds `finiteFuncCodeLevel` and that at least one argument is at `finiteFuncCodeLevel`. The two arguments are called `op1` and `op2`.

Following are the steps taken in `FiniteFuncNormalElement::doMultiply`. `s1` and `s2` are temporary variables.

1. If `op1` is finite return a copy of `op2` with its factor multiplied by `op1` and return that value.
2. If `op1` is at `cantorCodeLevel` assign to `s1` the `exponent` of `op1` otherwise assign to `s1` a copy of `op1` with `factor` set to 1.
3. If `op2` is at `cantorCodeLevel` assign to `s2` the `exponent` of `op2` otherwise assign to `s2` a copy of `op2` with `factor` set to 1.
4. Add `s1` and `s2` to create `newExp`.
5. If `newExp` has a single term and the `codeLevel` of that term is $\geq$ `finiteFuncCodeLevel` and the `factor` of that term is 1 then return the value of the single term of `newExp`.
6. Create and return a `CantorNormalElement` with exponent equal to `newExp` and `factor` equal to the `factor` or `op2`.

Some examples are shown in Table 14.

[32]A C++ `static` function is a member function of a `class` that is *not* associated with a particular instance of that `class`.

α	β	$\alpha \times \beta$
$\varepsilon_0 + 2$	$\varepsilon_0 + 2$	$\omega^{(\varepsilon_0 2)} + (\varepsilon_0 2) + 2$
$\varepsilon_0 + 2$	$\varepsilon_1 + \varepsilon_0 + 2$	$\varepsilon_1 + \omega^{(\varepsilon_0 2)} + (\varepsilon_0 2) + 2$
$\varepsilon_0 + 2$	$\omega^{\omega^{(\varepsilon_0 2)}}$	$\omega^{\omega^{(\varepsilon_0 2)}}$
$\varepsilon_0 + 2$	$\varphi(\omega^\omega, \omega + 3)$	$\varphi(\omega^\omega, \omega + 3)$
$\varepsilon_0 + 2$	$\varphi(3, \varepsilon_0, \varphi(\omega^\omega, \omega + 3))$	$\varphi(3, \varepsilon_0, \varphi(\omega^\omega, \omega + 3))$
$\varepsilon_1 + \varepsilon_0 + 2$	$\varepsilon_0 + 2$	$\omega^{\varepsilon_1 + \varepsilon_0} + (\varepsilon_1 2) + \varepsilon_0 + 2$
$\varepsilon_1 + \varepsilon_0 + 2$	$\varepsilon_1 + \varepsilon_0 + 2$	$\omega^{(\varepsilon_1 2)} + \omega^{\varepsilon_1 + \varepsilon_0} + (\varepsilon_1 2) + \varepsilon_0 + 2$
$\varepsilon_1 + \varepsilon_0 + 2$	$\omega^{\omega^{(\varepsilon_0 2)}}$	$\omega^{\varepsilon_1 + \omega^{(\varepsilon_0 2)}}$
$\varepsilon_1 + \varepsilon_0 + 2$	$\varphi(\omega^\omega, \omega + 3)$	$\varphi(\omega^\omega, \omega + 3)$
$\varepsilon_1 + \varepsilon_0 + 2$	$\varphi(3, \varepsilon_0, \varphi(\omega^\omega, \omega + 3))$	$\varphi(3, \varepsilon_0, \varphi(\omega^\omega, \omega + 3))$
$\omega^{\omega^{(\varepsilon_0 2)}}$	$\varepsilon_0 + 2$	$\omega^{\omega^{(\varepsilon_0 2)} + \varepsilon_0} + \omega^{\omega^{(\varepsilon_0 2)}} 2$
$\omega^{\omega^{(\varepsilon_0 2)}}$	$\varepsilon_1 + \varepsilon_0 + 2$	$\varepsilon_1 + \omega^{\omega^{(\varepsilon_0 2)} + \varepsilon_0} + \omega^{\omega^{(\varepsilon_0 2)}} 2$
$\omega^{\omega^{(\varepsilon_0 2)}}$	$\omega^{\omega^{(\varepsilon_0 2)}}$	$\omega^{\omega^{(\varepsilon_0 2)} 2}$
$\omega^{\omega^{(\varepsilon_0 2)}}$	$\varphi(\omega^\omega, \omega + 3)$	$\varphi(\omega^\omega, \omega + 3)$
$\omega^{\omega^{(\varepsilon_0 2)}}$	$\varphi(3, \varepsilon_0, \varphi(\omega^\omega, \omega + 3))$	$\varphi(3, \varepsilon_0, \varphi(\omega^\omega, \omega + 3))$
$\varphi(\omega^\omega, \omega + 3)$	$\varepsilon_0 + 2$	$\omega^{\varphi(\omega^\omega, \omega+3) + \varepsilon_0} + (\varphi(\omega^\omega, \omega + 3) 2)$
$\varphi(\omega^\omega, \omega + 3)$	$\varepsilon_1 + \varepsilon_0 + 2$	$\omega^{\varphi(\omega^\omega, \omega+3) + \varepsilon_1} + \omega^{\varphi(\omega^\omega, \omega+3) + \varepsilon_0} + (\varphi(\omega^\omega, \omega + 3) 2)$
$\varphi(\omega^\omega, \omega + 3)$	$\omega^{\omega^{(\varepsilon_0 2)}}$	$\omega^{\varphi(\omega^\omega, \omega+3) + \omega^{(\varepsilon_0 2)}}$
$\varphi(\omega^\omega, \omega + 3)$	$\varphi(\omega^\omega, \omega + 3)$	$\omega^{(\varphi(\omega^\omega, \omega+3) 2)}$
$\varphi(\omega^\omega, \omega + 3)$	$\varphi(3, \varepsilon_0, \varphi(\omega^\omega, \omega + 3))$	$\varphi(3, \varepsilon_0, \varphi(\omega^\omega, \omega + 3))$
$\varphi(3, \varepsilon_0, \varphi(\omega^\omega, \omega + 3))$	$\varepsilon_0 + 2$	$\omega^{\varphi(3, \varepsilon_0, \varphi(\omega^\omega, \omega+3)) + \varepsilon_0} + (\varphi(3, \varepsilon_0, \varphi(\omega^\omega, \omega + 3)) 2)$
$\varphi(3, \varepsilon_0, \varphi(\omega^\omega, \omega + 3))$	$\varepsilon_1 + \varepsilon_0 + 2$	$\omega^{\varphi(3, \varepsilon_0, \varphi(\omega^\omega, \omega+3)) + \varepsilon_1} + \omega^{\varphi(3, \varepsilon_0, \varphi(\omega^\omega, \omega+3)) + \varepsilon_0} + (\varphi(3, \varepsilon_0, \varphi(\omega^\omega, \omega + 3)) 2)$
$\varphi(3, \varepsilon_0, \varphi(\omega^\omega, \omega + 3))$	$\omega^{\omega^{(\varepsilon_0 2)}}$	$\omega^{\varphi(3, \varepsilon_0, \varphi(\omega^\omega, \omega+3)) + \omega^{(\varepsilon_0 2)}}$
$\varphi(3, \varepsilon_0, \varphi(\omega^\omega, \omega + 3))$	$\varphi(\omega^\omega, \omega + 3)$	$\omega^{\varphi(3, \varepsilon_0, \varphi(\omega^\omega, \omega+3)) + \varphi(\omega^\omega, \omega+3)}$
$\varphi(3, \varepsilon_0, \varphi(\omega^\omega, \omega + 3))$	$\varphi(3, \varepsilon_0, \varphi(\omega^\omega, \omega + 3))$	$\omega^{(\varphi(3, \varepsilon_0, \varphi(\omega^\omega, \omega+3)) 2)}$

Table 14: `FiniteFuncOrdinal` multiply examples

6.4.2 `FiniteFuncOrdinal` exponentiation

Exponentiation has a structure similar to multiplication. The only routines that need to be overridden involve a^b when both a and b are single terms in a normal form expansion. The routines overridden are `toPower` and `powerOf`. Most of the work is done by `static` member function `FiniteFuncNormalElement::doToPower` which takes both parameters as arguments. The two routines that call this only check if both operands are at `cantorCodeLevel` and if so call the corresponding `CantorNormalElement` member function.

The key to the algorithm is again the observation that every normal form term at `finiteFuncCodeLevel` with a `factor` of 1 is a fixed point of ω^x, i. e. $a = \omega^a$. The value of `factor` can be ignored in the base part of an exponential expression where the exponent is a limit ordinal. All infinite normal form terms are limit ordinals. Thus a^b where both a and b are at `finiteFuncCodeLevel` and $b \leq a$ is ω^{ab} or equivalently $\omega^{\omega^{a+b}}$ which is the normal form result. If the base, a, of the exponential is at `cantorCodeLevel` then $a = \omega^\alpha$ and the result is $\omega^{\alpha b}$.

Following is a more detailed summary of `FiniteFuncNormalElement::doToPower`. This summary describes how $\texttt{base}^{\texttt{expon}}$ is computed.

- If ($\texttt{base} < \texttt{expon}$) $\wedge$ ($\texttt{expon.factor} = 1$) $\wedge$ ($\texttt{expon.expTerm()}$[33] `== true`) then `expon` is returned.
- $\texttt{p1} = \texttt{base.codeLevel} \geq \texttt{finiteFuncCodeLevel}\ ?\texttt{base} : \texttt{base.exponent}$[34]
- $\texttt{newExp} = \texttt{p1} \times \texttt{expon}$
- If `newExp` has a single term with a `factor` of 1 and the `codeLevel` of that term is $\geq$ `finiteFuncCodeLevel` then return `newExp` because it is a fixed point of ω^x.
- Otherwise return $\omega^{\texttt{newExp}}$.

[33]`CantorNormalElement::expTerm` returns `true` iff the term it is called from is at `cantorCodeLevel`, has a factor or 1, has an `exponent` with a single term and such that `exponent.expTerm()` returns `true`.

[34]In C++ ‘boolean expression ? optionTrue : optionFalse’ evaluates to ‘optionTrue’ if ‘boolean expression’ is true and ‘optionFalse’ otherwise

α	β	α^{β}
ε_0+2	ε_0+2	$\omega^{\omega^{(\varepsilon_0 2)}+(\varepsilon_0 2)}+\omega^{\omega^{(\varepsilon_0 2)}+\varepsilon_0}2+\omega^{\omega^{(\varepsilon_0 2)}}2$
ε_0+2	$\varepsilon_1+\varepsilon_0$	$\omega^{\varepsilon_1+\omega^{(\varepsilon_0 2)}}$
ε_0+2	$\omega^{\omega^{(\varepsilon_0 2)}}+3$	$\omega^{\omega^{\omega^{(\varepsilon_0 2)}}+(\varepsilon_0 3)}+\omega^{\omega^{\omega^{(\varepsilon_0 2)}}+(\varepsilon_0 2)}2+\omega^{\omega^{\omega^{(\varepsilon_0 2)}}+\varepsilon_0}2+\omega^{\omega^{\omega^{(\varepsilon_0 2)}}}2$
ε_0+2	$\varphi(\omega^{\omega},\omega+3)$	$\varphi(\omega^{\omega},\omega+3)$
ε_0+2	$\varphi(3,\varepsilon_0,\varphi(\omega^{\omega},\omega+3))$	$\varphi(3,\varepsilon_0,\varphi(\omega^{\omega},\omega+3))$
$\varepsilon_1+\varepsilon_0$	ε_0+2	$\omega^{\omega^{\varepsilon_1+\varepsilon_0}+(\varepsilon_1 2)}+\omega^{\omega^{\varepsilon_1+\varepsilon_0}+\varepsilon_1+\varepsilon_0}$
$\varepsilon_1+\varepsilon_0$	$\varepsilon_1+\varepsilon_0$	$\omega^{\omega^{(\varepsilon_1 2)}+\omega^{\varepsilon_1+\varepsilon_0}}$
$\varepsilon_1+\varepsilon_0$	$\omega^{\omega^{(\varepsilon_0 2)}}+3$	$\omega^{\omega^{\varepsilon_1+\omega^{(\varepsilon_0 2)}}+(\varepsilon_1 3)}+\omega^{\omega^{\varepsilon_1+\omega^{(\varepsilon_0 2)}}+(\varepsilon_1 2)+\varepsilon_0}$
$\varepsilon_1+\varepsilon_0$	$\varphi(\omega^{\omega},\omega+3)$	$\varphi(\omega^{\omega},\omega+3)$
$\varepsilon_1+\varepsilon_0$	$\varphi(3,\varepsilon_0,\varphi(\omega^{\omega},\omega+3))$	$\varphi(3,\varepsilon_0,\varphi(\omega^{\omega},\omega+3))$
$\omega^{\omega^{(\varepsilon_0 2)}}+3$	ε_0+2	$\omega^{\omega^{(\varepsilon_0 3)}+\omega^{(\varepsilon_0 2)}2}+\omega^{\omega^{(\varepsilon_0 3)}+\omega^{(\varepsilon_0 2)}}3+\omega^{\omega^{(\varepsilon_0 3)}}3$
$\omega^{\omega^{(\varepsilon_0 2)}}+3$	$\varepsilon_1+\varepsilon_0$	$\omega^{\varepsilon_1+\omega^{(\varepsilon_0 3)}}$
$\omega^{\omega^{(\varepsilon_0 2)}}+3$	$\omega^{\omega^{(\varepsilon_0 2)}}+3$	$\omega^{\omega^{\omega^{(\varepsilon_0 2)}}+\omega^{(\varepsilon_0 2)}3}+\omega^{\omega^{\omega^{(\varepsilon_0 2)}}+\omega^{(\varepsilon_0 2)}2}3+\omega^{\omega^{\omega^{(\varepsilon_0 2)}}+\omega^{(\varepsilon_0 2)}}3+\omega^{\omega^{\omega^{(\varepsilon_0 2)}}}3$
$\omega^{\omega^{(\varepsilon_0 2)}}+3$	$\varphi(\omega^{\omega},\omega+3)$	$\varphi(\omega^{\omega},\omega+3)$
$\omega^{\omega^{(\varepsilon_0 2)}}+3$	$\varphi(3,\varepsilon_0,\varphi(\omega^{\omega},\omega+3))$	$\varphi(3,\varepsilon_0,\varphi(\omega^{\omega},\omega+3))$
$\varphi(\omega^{\omega},\omega+3)$	ε_0+2	$\omega^{\omega^{\varphi(\omega^{\omega},\omega+3)+\varepsilon_0}+(\varphi(\omega^{\omega},\omega+3)2)}$
$\varphi(\omega^{\omega},\omega+3)$	$\varepsilon_1+\varepsilon_0$	$\omega^{\omega^{\varphi(\omega^{\omega},\omega+3)+\varepsilon_1}+\omega^{\varphi(\omega^{\omega},\omega+3)+\varepsilon_0}}$
$\varphi(\omega^{\omega},\omega+3)$	$\omega^{\omega^{(\varepsilon_0 2)}}+3$	$\omega^{\omega^{\varphi(\omega^{\omega},\omega+3)+\omega^{(\varepsilon_0 2)}}+(\varphi(\omega^{\omega},\omega+3)3)}$
$\varphi(\omega^{\omega},\omega+3)$	$\varphi(\omega^{\omega},\omega+3)$	$\omega^{\omega^{(\varphi(\omega^{\omega},\omega+3)2)}}$
$\varphi(\omega^{\omega},\omega+3)$	$\varphi(3,\varepsilon_0,\varphi(\omega^{\omega},\omega+3))$	$\varphi(3,\varepsilon_0,\varphi(\omega^{\omega},\omega+3))$
$\varphi(3,\varepsilon_0,\varphi(\omega^{\omega},\omega+3))$	ε_0+2	$\omega^{\omega^{\varphi(3,\varepsilon_0,\varphi(\omega^{\omega},\omega+3))+\varepsilon_0}+(\varphi(3,\varepsilon_0,\varphi(\omega^{\omega},\omega+3))2)}$
$\varphi(3,\varepsilon_0,\varphi(\omega^{\omega},\omega+3))$	$\varepsilon_1+\varepsilon_0$	$\omega^{\omega^{\varphi(3,\varepsilon_0,\varphi(\omega^{\omega},\omega+3))+\varepsilon_1}+\omega^{\varphi(3,\varepsilon_0,\varphi(\omega^{\omega},\omega+3))+\varepsilon_0}}$
$\varphi(3,\varepsilon_0,\varphi(\omega^{\omega},\omega+3))$	$\omega^{\omega^{(\varepsilon_0 2)}}+3$	$\omega^{\omega^{\varphi(3,\varepsilon_0,\varphi(\omega^{\omega},\omega+3))+\omega^{(\varepsilon_0 2)}}+(\varphi(3,\varepsilon_0,\varphi(\omega^{\omega},\omega+3))3)}$
$\varphi(3,\varepsilon_0,\varphi(\omega^{\omega},\omega+3))$	$\varphi(\omega^{\omega},\omega+3)$	$\omega^{\omega^{\varphi(3,\varepsilon_0,\varphi(\omega^{\omega},\omega+3))+\varphi(\omega^{\omega},\omega+3)}}$
$\varphi(3,\varepsilon_0,\varphi(\omega^{\omega},\omega+3))$	$\varphi(3,\varepsilon_0,\varphi(\omega^{\omega},\omega+3))$	$\omega^{\omega^{(\varphi(3,\varepsilon_0,\varphi(\omega^{\omega},\omega+3))2)}}$

Table 15: `FiniteFuncOrdinal` exponential examples

'cp' stands for function `createParameters`	
C++ code	**Ordinal**
`iterativeFunctional(zero,cp(&one))`	ω
`iterativeFunctional(zero,cp(&one,&zero))`	ε_0
`iterativeFunctional(zero,cp(&one,&one,&zero))`	Γ_1
`iterativeFunctional(one)`	φ_1
`iterativeFunctional(one,cp(&one))`	$\varphi_1(1)$
`iterativeFunctional(one,cp(&one,&omega))`	$\varphi_1(1,\omega)$
`iterativeFunctional(one,cp(&one,&omega,&zero,&zero))`	$\varphi_1(1,\omega,0,0)$
`iterativeFunctional(omega,cp(&one,&omega,&zero,&zero))`	$\varphi_\omega(1,\omega,0,0)$
`iterativeFunctional(omega)`	φ_ω
`iterativeFunctional(one,cp(&iterativeFunctional(omega)))`	φ_ω

Table 16: `iterativeFunctional` C++ code examples

7 `IterFuncOrdinal` class

C++ `class IterFuncOrdinal` is derived from `class FiniteFuncOrdinal` which in turn is derived from `class Ordinal`. It implements the iterative functional hierarchy described in Section 5.5. It uses the normal form in Equation 7. Each term of that normal form, each $\alpha_i n_i$, is represented by an instance of `class IterFuncNormalElement` or one of the `class`es this class is derived from. These are `FiniteFuncNormalElement` and `CantorNormalElement`. Terms that are `IterFuncNormalElement`s have a `codeLevel` of `iterFuncCodeLevel`.

The `IterFuncOrdinal class` should not be used directly to create ordinal notations. Instead use function `iterativeFunctional`[35]. This function takes two arguments. The first gives the level of iteration or the value of γ_i in Equation 7. The second gives a `NULL` terminated array of pointers to `Ordinals` which are the $\beta_{i,j}$ in Equation 7. This second parameter is optional and can be created with function `createParameters` described in Section 6. Some examples are shown in Table 16.

`iterativeFunctional` creates an `Ordinal` or `FiniteFuncOrdinal` instead of an `IterFuncOrdinal` if appropriate. The first three lines of Table 16 are examples. It also reduces fixed points to their simplest possible expression. The last line of the table is an example.

7.1 `IterFuncNormalElement::compare` member function

`IterFuncNormalElement::compare` supports one term in the extended normal form in Equation 7. `IterFuncNormalElement::compare` with a single `CantorNormalElement` argument overrides `FiniteFuncNormalElement::compare` with the same argument (see Section 6.1). It outputs 1, 0 or -1 if the object it is called from is greater than equal to or less than its argument term.

[35] The interactive ordinal calculator supports a TeX like syntax. To define $\varphi_a(b,c,d)$ write `psi_{a}(b,c,d)`. Any number of parameters in parenthesis may be entered or the parenthesis may be omitted. See Section A.11.6 for examples.

The `IterFuncNormalElement` object `compare` (or any `class` member function) is called from is `this` in C++. The `CantorNormalElement` argument to `compare` is `trm`. If `trm.codeLevel` $>$ `iterFuncCodeLevel` then `-trm.compare(*this)` is returned. This invokes the subclass member function associated with the `codeLevel` of `trm`.

This `compare` is similar to that for `class FiniteFuncOrdinal` described in Section 6.1. The main difference is additional tests on γ_i from Equation 7.

If `trm.codeLevel` $<$ `iterFuncCodeLevel` then `trm` $>$ `this` only if the maximum parameter of `trm` is greater than `this`. However the values of `factor` must be ignored in making this comparison because `trm` $\geq \omega^{\texttt{trm.getMaxParameter()}}$ and this will swamp the effect of any finite `factor`.

Following is an outline of `IterFuncNormalElement::compare` with a `CantorNormalElement` parameter `trm`.

1. If `trm.codeLevel` $<$ `iterFuncCodeLevel`, `this` is compared with the first (largest) term of the maximum parameter of the argument (ignoring the two `factor`s). If the result is ≤ 0, -1 is returned. Otherwise 1 is returned.

2. `this` is compared to the maximum parameter of the argument. If the result is less than than or equal -1 is returned.

3. The maximum parameter of `this` is compared against the argument `trm`. If the result is greater or equal 1 is returned.

4. The function level (`functionLevel`) (γ_i from Equation 7) of `this` is compared to the `functionLevel` of `trm`. If the result is nonzero it is returned.

If no result is obtained `IterFuncNormalElement::compareFiniteParams` is called to compare in sequence the number of parameters of the two terms and then the size of each argument in succession starting with the most significant. If any difference is encountered that is returned as the result. If not the difference in the `factor`s of the two terms is returned.

7.2 `IterFuncNormalElement::limitElement` member function

`IterFuncNormalElement::limitElement` overrides `CantorNormalElement::limitElement` described in Section 4.3 and `FiniteFuncNormalElement::limitElement` described in Section 6.2 This function takes a single integer parameter. Increasing values for this argument yield larger ordinal notations as output. The union of the ordinals represented by the outputs for all integer inputs is equal to the ordinal represented by the `IterFuncNormalElement` `class` instance `limitElement` is called from. This will be referred to as the input to `limitElement`.

`Ordinal::limitElement` processes all but the last term of the normal form of the result by copying it unchanged from the input `Ordinal`. The last term of the result is determined by a number of conditions on the last term of the input. This is what `IterFuncNormalElement::limitElement` does.

Tables 2, 9 and 10 fully define `IterFuncOrdinal::limitElement`.[36] The C++ pseudo code in the table uses shorter variable names and takes other shortcuts, but accurately reflects the logic in the source code. The `IterFuncNormalElement` version of this routine calls a portion of the `FiniteFuncNormalElement` version called `limitElementCom`. `FiniteFuncNormalElement::limitElementCom` always creates its return value with a `virtual` function `createVirtualOrdImpl` which is overrides when it is called from a subclass object. The `rep1` and `rep2`[37] of tables 9 and 10 also always call this `virtual` function to create a result.

Some examples are shown in Table 17.

[36]The '**X**' column in Tables 9 and 10 connect each table entry to the section of code preceding `RETURN1`. This is a debugging macro which has a quoted string as a parameter. This string is an exit code that matches the **X** column in the tables. If debugging mode is enabled for `compare` these exit codes are displayed.

[37]The name of these functions in the C++ source are `replace1` and `replace2`.

`IterFuncOrdinal`	`limitElement`		
ω	1	2	3
φ_1	ω	ε_0	Γ_0
φ_3	$\varphi_2(\varphi_2+1)$	$\varphi_2(\varphi_2+1,0)$	$\varphi_2(\varphi_2+1,0,0)$
$\varphi_1(1)$	ω^{φ_1+1}	$\varphi(\varphi_1+1,0)$	$\varphi(\varphi_1+1,0,0)$
φ_ω	φ_1	φ_2	φ_3
$\varphi_\omega(1)$	$\varphi_1(\varphi_\omega+1)$	$\varphi_2(\varphi_\omega+1)$	$\varphi_3(\varphi_\omega+1)$
$\varphi_1(2)$	$\omega^{\varphi_1(1)+1}$	$\varphi(\varphi_1(1)+1,0)$	$\varphi(\varphi_1(1)+1,0,0)$
$\varphi_1(\varepsilon_0)$	$\varphi_1(\omega)$	$\varphi_1(\omega^\omega)$	$\varphi_1(\omega^{\omega^\omega})$
$\varphi_1(1,0,0)$	$\varphi_1(1,0)$	$\varphi_1(\varphi_1(1,0)+1,0)$	$\varphi_1(\varphi_1(\varphi_1(1,0)+1,0)+1,0)$
$\varphi_3(1,0,0)$	$\varphi_3(1,0)$	$\varphi_3(\varphi_3(1,0)+1,0)$	$\varphi_3(\varphi_3(\varphi_3(1,0)+1,0)+1,0)$
$\varphi_1(\omega,1)$	$\varphi_1(1,\varphi_1(\omega,0)+1)$	$\varphi_1(2,\varphi_1(\omega,0)+1)$	$\varphi_1(3,\varphi_1(\omega,0)+1)$
$\varphi_1(\omega,1,0)$	$\varphi_1(\omega,0,1)$	$\varphi_1(\omega,0,\varphi_1(\omega,0,1)+1)$	$\varphi_1(\omega,0,\varphi_1(\omega,0,\varphi_1(\omega,0,1)+1)+1)$
φ_2	$\varphi_1(\varphi_1+1)$	$\varphi_1(\varphi_1+1,0)$	$\varphi_1(\varphi_1+1,0,0)$
φ_{ε_0}	φ_ω	φ_{ω^ω}	$\varphi_{\omega^{\omega^\omega}}$
φ_{Γ_0}	$\varphi_{\varepsilon_0+1}$	$\varphi_{\varphi(\varepsilon_0+1,0)+1}$	$\varphi_{\varphi(\varphi(\varepsilon_0+1,0)+1,0)+1}$
$\varphi_{\Gamma_0}(1)$	$\varphi_{\varepsilon_0+1}(\varphi_{\Gamma_0}+1)$	$\varphi_{\varphi(\varepsilon_0+1,0)+1}(\varphi_{\Gamma_0}+1)$	$\varphi_{\varphi(\varphi(\varepsilon_0+1,0)+1,0)+1}(\varphi_{\Gamma_0}+1)$
$\varphi_{\Gamma_0}(1,0)$	$\varphi_{\Gamma_0}(1)$	$\varphi_{\Gamma_0}(\varphi_{\Gamma_0}(1)+1)$	$\varphi_{\Gamma_0}(\varphi_{\Gamma_0}(\varphi_{\Gamma_0}(1)+1)+1)$
$\varphi_{\Gamma_0}(\varepsilon_0)$	$\varphi_{\Gamma_0}(\omega)$	$\varphi_{\Gamma_0}(\omega^\omega)$	$\varphi_{\Gamma_0}(\omega^{\omega^\omega})$
$\varphi_{\Gamma_0}(\varepsilon_0,0)$	$\varphi_{\Gamma_0}(\omega,0)$	$\varphi_{\Gamma_0}(\omega^\omega,0)$	$\varphi_{\Gamma_0}(\omega^{\omega^\omega},0)$
$\varphi_{\Gamma_0}(\varepsilon_0,1)$	$\varphi_{\Gamma_0}(\omega,\varphi_{\Gamma_0}(\varepsilon_0,0)+1)$	$\varphi_{\Gamma_0}(\omega^\omega,\varphi_{\Gamma_0}(\varepsilon_0,0)+1)$	$\varphi_{\Gamma_0}(\omega^{\omega^\omega},\varphi_{\Gamma_0}(\varepsilon_0,0)+1)$
$\varphi_{\Gamma_0}(\varepsilon_0,1,1)$	$\varphi_{\Gamma_0}(\varepsilon_0,1,0)+1$	$\varphi_{\Gamma_0}(\varepsilon_0,0,\varphi_{\Gamma_0}(\varepsilon_0,1,0)+1)$	$\varphi_{\Gamma_0}(\varepsilon_0,0,\varphi_{\Gamma_0}(\varepsilon_0,0,\varphi_{\Gamma_0}(\varepsilon_0,1,0)+1)+1)$
$\varphi_{\Gamma_0}(\varepsilon_0,1,1,2)$	$\varphi_{\Gamma_0}(\varepsilon_0,1,1,1)+1$	$\varphi_{\Gamma_0}(\varepsilon_0,1,0,\varphi_{\Gamma_0}(\varepsilon_0,1,1,1)+1)$	$\varphi_{\Gamma_0}(\varepsilon_0,1,0,\varphi_{\Gamma_0}(\varepsilon_0,1,0,\varphi_{\Gamma_0}(\varepsilon_0,1,1,1)+1)+1)$

Table 17: `IterFuncOrdinal::limitElement` examples.

7.3 `IterFuncOrdinal::fixedPoint` member function

`IterFuncOrdinal::fixedPoint` is used by `iterativeFunctional` to create an instance of an `IterFuncOrdinal` in a normal form (Equation 7) that is the simplest expression for the ordinal represented. The routine has the following parameters for a single term in Equation 7.

- The function level, γ.
- An index specifying the largest parameter of the ordinal notation being constructed. If the largest parameter is the function level the index has the value `iterMaxParam` defined as -1 in an `enum`.
- The function parameters as an array of pointers to `Ordinals`. These are the β_j in a normal form term.

  in Equation 7.

This function determines if the parameter at the specified index is a fixed point for an `IterFuncOrdinal` created with the specified parameters. If it is, `true` is returned and otherwise `false`. The routine that calls this routine selects the largest parameter from the function level (γ) and the array of `Ordinal` pointers (β_j) as the one to check and indicates this in the index parameter, The calling routine checks to see if all less significant parameters are 0. If not this cannot be a fixed point. Thus `fixedPoint` is called only if this condition is met.

Section 6.3 describes `psuedoCodeLevel`. If the `psuedoCodeLevel` of the selected parameter is less than or equal `cantorCodeLevel`, `false` is returned. If that level is greater than `iterFuncCodeLevel`, `true` is returned. The most significant parameter, the function level, cannot be a fixed point unless it has a `psuedoCodeLevel` $>$ `iterFuncCodeLevel`. Thus, if the index selects the the function level, and the previous test was not passed `false` is returned. Finally an `IterFuncOrdinal` is constructed from all the parameters except that selected by the index. If this value is less than the selected parameter, `true` is returned and otherwise `false`.

7.4 `IterFuncOrdinal` operators

The multiply and exponential routines for `FiniteFuncOrdinal` and `Ordinal` and the `classes` for normal form terms `CantorNormalElement` and `FiniteFuncNormalElement` do not have functions that need to be overridden to support `IterFuncOrdinal` multiplication and exponentiation. The exceptions are utilities such as that used to create a copy of normal form term `IterFuncNormalElement` with a new value for `factor`.

Some multiply examples are shown in Table 18. Some exponential examples are shown in Table 19.

α	β	$\alpha \times \beta$
φ_1	φ_1	$\omega^{(\varphi_1 2)}$
φ_1	φ_3	φ_3
φ_1	$\varphi_3(1,0,0)$	$\varphi_3(1,0,0)$
φ_1	$\varphi_1(\omega,1)$	$\varphi_1(\omega,1)$
φ_1	$\varphi_1(\omega,1,0)+\varphi_1$	$\varphi_1(\omega,1,0)+\omega^{(\varphi_1 2)}$
φ_1	$\varphi_{\varepsilon_0}+\varphi_1(\omega,1,0)+\varphi_1$	$\varphi_{\varepsilon_0}+\varphi_1(\omega,1,0)+\omega^{(\varphi_1 2)}$
φ_3	φ_1	$\omega^{\varphi_3+\varphi_1}$
φ_3	φ_3	$\omega^{(\varphi_3 2)}$
φ_3	$\varphi_3(1,0,0)$	$\varphi_3(1,0,0)$
φ_3	$\varphi_1(\omega,1)$	$\omega^{\varphi_3+\varphi_1(\omega,1)}$
φ_3	$\varphi_1(\omega,1,0)+\varphi_1$	$\omega^{\varphi_3+\varphi_1(\omega,1,0)}+\omega^{\varphi_3+\varphi_1}$
φ_3	$\varphi_{\varepsilon_0}+\varphi_1(\omega,1,0)+\varphi_1$	$\varphi_{\varepsilon_0}+\omega^{\varphi_3+\varphi_1(\omega,1,0)}+\omega^{\varphi_3+\varphi_1}$
$\varphi_3(1,0,0)$	φ_1	$\omega^{\varphi_3(1,0,0)+\varphi_1}$
$\varphi_3(1,0,0)$	φ_3	$\omega^{\varphi_3(1,0,0)+\varphi_3}$
$\varphi_3(1,0,0)$	$\varphi_3(1,0,0)$	$\omega^{(\varphi_3(1,0,0)2)}$
$\varphi_3(1,0,0)$	$\varphi_1(\omega,1)$	$\omega^{\varphi_3(1,0,0)+\varphi_1(\omega,1)}$
$\varphi_3(1,0,0)$	$\varphi_1(\omega,1,0)+\varphi_1$	$\omega^{\varphi_3(1,0,0)+\varphi_1(\omega,1,0)}+\omega^{\varphi_3(1,0,0)+\varphi_1}$
$\varphi_3(1,0,0)$	$\varphi_{\varepsilon_0}+\varphi_1(\omega,1,0)+\varphi_1$	$\varphi_{\varepsilon_0}+\omega^{\varphi_3(1,0,0)+\varphi_1(\omega,1,0)}+\omega^{\varphi_3(1,0,0)+\varphi_1}$
$\varphi_1(\omega,1)$	φ_1	$\omega^{\varphi_1(\omega,1)+\varphi_1}$
$\varphi_1(\omega,1)$	φ_3	φ_3
$\varphi_1(\omega,1)$	$\varphi_3(1,0,0)$	$\varphi_3(1,0,0)$
$\varphi_1(\omega,1)$	$\varphi_1(\omega,1)$	$\omega^{(\varphi_1(\omega,1)2)}$
$\varphi_1(\omega,1)$	$\varphi_1(\omega,1,0)+\varphi_1$	$\varphi_1(\omega,1,0)+\omega^{\varphi_1(\omega,1)+\varphi_1}$
$\varphi_1(\omega,1)$	$\varphi_{\varepsilon_0}+\varphi_1(\omega,1,0)+\varphi_1$	$\varphi_{\varepsilon_0}+\varphi_1(\omega,1,0)+\omega^{\varphi_1(\omega,1)+\varphi_1}$
$\varphi_1(\omega,1,0)+\varphi_1$	φ_1	$\omega^{\varphi_1(\omega,1,0)+\varphi_1}$
$\varphi_1(\omega,1,0)+\varphi_1$	φ_3	φ_3
$\varphi_1(\omega,1,0)+\varphi_1$	$\varphi_3(1,0,0)$	$\varphi_3(1,0,0)$
$\varphi_1(\omega,1,0)+\varphi_1$	$\varphi_1(\omega,1)$	$\omega^{\varphi_1(\omega,1,0)+\varphi_1(\omega,1)}$
$\varphi_1(\omega,1,0)+\varphi_1$	$\varphi_1(\omega,1,0)+\varphi_1$	$\omega^{(\varphi_1(\omega,1,0)2)}+\omega^{\varphi_1(\omega,1,0)+\varphi_1}$
$\varphi_1(\omega,1,0)+\varphi_1$	$\varphi_{\varepsilon_0}+\varphi_1(\omega,1,0)+\varphi_1$	$\varphi_{\varepsilon_0}+\omega^{(\varphi_1(\omega,1,0)2)}+\omega^{\varphi_1(\omega,1,0)+\varphi_1}$
$\varphi_{\varepsilon_0}+\varphi_1(\omega,1,0)+\varphi_1$	φ_1	$\omega^{\varphi_{\varepsilon_0}+\varphi_1}$
$\varphi_{\varepsilon_0}+\varphi_1(\omega,1,0)+\varphi_1$	φ_3	$\omega^{\varphi_{\varepsilon_0}+\varphi_3}$
$\varphi_{\varepsilon_0}+\varphi_1(\omega,1,0)+\varphi_1$	$\varphi_3(1,0,0)$	$\omega^{\varphi_{\varepsilon_0}+\varphi_3(1,0,0)}$
$\varphi_{\varepsilon_0}+\varphi_1(\omega,1,0)+\varphi_1$	$\varphi_1(\omega,1)$	$\omega^{\varphi_{\varepsilon_0}+\varphi_1(\omega,1)}$
$\varphi_{\varepsilon_0}+\varphi_1(\omega,1,0)+\varphi_1$	$\varphi_1(\omega,1,0)+\varphi_1$	$\omega^{\varphi_{\varepsilon_0}+\varphi_1(\omega,1,0)}+\omega^{\varphi_{\varepsilon_0}+\varphi_1}$
$\varphi_{\varepsilon_0}+\varphi_1(\omega,1,0)+\varphi_1$	$\varphi_{\varepsilon_0}+\varphi_1(\omega,1,0)+\varphi_1$	$\omega^{(\varphi_{\varepsilon_0}2)}+\omega^{\varphi_{\varepsilon_0}+\varphi_1(\omega,1,0)}+\omega^{\varphi_{\varepsilon_0}+\varphi_1}$

Table 18: `IterFuncOrdinal` multiply examples

α	β	α^{β}
φ_1	φ_1	$\omega^{\omega^{(\varphi_1 2)}}$
φ_1	φ_3	φ_3
φ_1	$\varphi_3(1,0,0)$	$\varphi_3(1,0,0)$
φ_1	$\varphi_1(\omega,1)$	$\varphi_1(\omega,1)$
φ_1	$\varphi_1(\omega,1,0)+\varphi_1$	$\omega^{\varphi_1(\omega,1,0)+\omega^{(\varphi_1 2)}}$
φ_1	$\varphi_{\varepsilon_0}+\varphi_1(\omega,1,0)+\varphi_1$	$\omega^{\varphi_{\varepsilon_0}+\varphi_1(\omega,1,0)+\omega^{(\varphi_1 2)}}$
φ_3	φ_1	$\omega^{\omega^{\varphi_3+\varphi_1}}$
φ_3	φ_3	$\omega^{\omega^{(\varphi_3 2)}}$
φ_3	$\varphi_3(1,0,0)$	$\varphi_3(1,0,0)$
φ_3	$\varphi_1(\omega,1)$	$\omega^{\omega^{\varphi_3+\varphi_1(\omega,1)}}$
φ_3	$\varphi_1(\omega,1,0)+\varphi_1$	$\omega^{\omega^{\varphi_3+\varphi_1(\omega,1,0)}+\omega^{\varphi_3+\varphi_1}}$
φ_3	$\varphi_{\varepsilon_0}+\varphi_1(\omega,1,0)+\varphi_1$	$\omega^{\varphi_{\varepsilon_0}+\omega^{\varphi_3+\varphi_1(\omega,1,0)}+\omega^{\varphi_3+\varphi_1}}$
$\varphi_3(1,0,0)$	φ_1	$\omega^{\omega^{\varphi_3(1,0,0)+\varphi_1}}$
$\varphi_3(1,0,0)$	φ_3	$\omega^{\omega^{\varphi_3(1,0,0)+\varphi_3}}$
$\varphi_3(1,0,0)$	$\varphi_3(1,0,0)$	$\omega^{\omega^{(\varphi_3(1,0,0)2)}}$
$\varphi_3(1,0,0)$	$\varphi_1(\omega,1)$	$\omega^{\omega^{\varphi_3(1,0,0)+\varphi_1(\omega,1)}}$
$\varphi_3(1,0,0)$	$\varphi_1(\omega,1,0)+\varphi_1$	$\omega^{\omega^{\varphi_3(1,0,0)+\varphi_1(\omega,1,0)}+\omega^{\varphi_3(1,0,0)+\varphi_1}}$
$\varphi_3(1,0,0)$	$\varphi_{\varepsilon_0}+\varphi_1(\omega,1,0)+\varphi_1$	$\omega^{\varphi_{\varepsilon_0}+\omega^{\varphi_3(1,0,0)+\varphi_1(\omega,1,0)}+\omega^{\varphi_3(1,0,0)+\varphi_1}}$
$\varphi_1(\omega,1)$	φ_1	$\omega^{\omega^{\varphi_1(\omega,1)+\varphi_1}}$
$\varphi_1(\omega,1)$	φ_3	φ_3
$\varphi_1(\omega,1)$	$\varphi_3(1,0,0)$	$\varphi_3(1,0,0)$
$\varphi_1(\omega,1)$	$\varphi_1(\omega,1)$	$\omega^{\omega^{(\varphi_1(\omega,1)2)}}$
$\varphi_1(\omega,1)$	$\varphi_1(\omega,1,0)+\varphi_1$	$\omega^{\varphi_1(\omega,1,0)+\omega^{\varphi_1(\omega,1)+\varphi_1}}$
$\varphi_1(\omega,1)$	$\varphi_{\varepsilon_0}+\varphi_1(\omega,1,0)+\varphi_1$	$\omega^{\varphi_{\varepsilon_0}+\varphi_1(\omega,1,0)+\omega^{\varphi_1(\omega,1)+\varphi_1}}$
$\varphi_1(\omega,1,0)+\varphi_1$	φ_1	$\omega^{\omega^{\varphi_1(\omega,1,0)+\varphi_1}}$
$\varphi_1(\omega,1,0)+\varphi_1$	φ_3	φ_3
$\varphi_1(\omega,1,0)+\varphi_1$	$\varphi_3(1,0,0)$	$\varphi_3(1,0,0)$
$\varphi_1(\omega,1,0)+\varphi_1$	$\varphi_1(\omega,1)$	$\omega^{\omega^{\varphi_1(\omega,1,0)+\varphi_1(\omega,1)}}$
$\varphi_1(\omega,1,0)+\varphi_1$	$\varphi_1(\omega,1,0)+\varphi_1$	$\omega^{\omega^{(\varphi_1(\omega,1,0)2)}+\omega^{\varphi_1(\omega,1,0)+\varphi_1}}$
$\varphi_1(\omega,1,0)+\varphi_1$	$\varphi_{\varepsilon_0}+\varphi_1(\omega,1,0)+\varphi_1$	$\omega^{\varphi_{\varepsilon_0}+\omega^{(\varphi_1(\omega,1,0)2)}+\omega^{\varphi_1(\omega,1,0)+\varphi_1}}$
$\varphi_{\varepsilon_0}+\varphi_1(\omega,1,0)+\varphi_1$	φ_1	$\omega^{\omega^{\varphi_{\varepsilon_0}+\varphi_1}}$
$\varphi_{\varepsilon_0}+\varphi_1(\omega,1,0)+\varphi_1$	φ_3	$\omega^{\omega^{\varphi_{\varepsilon_0}+\varphi_3}}$
$\varphi_{\varepsilon_0}+\varphi_1(\omega,1,0)+\varphi_1$	$\varphi_3(1,0,0)$	$\omega^{\omega^{\varphi_{\varepsilon_0}+\varphi_3(1,0,0)}}$
$\varphi_{\varepsilon_0}+\varphi_1(\omega,1,0)+\varphi_1$	$\varphi_1(\omega,1)$	$\omega^{\omega^{\varphi_{\varepsilon_0}+\varphi_1(\omega,1)}}$
$\varphi_{\varepsilon_0}+\varphi_1(\omega,1,0)+\varphi_1$	$\varphi_1(\omega,1,0)+\varphi_1$	$\omega^{\omega^{\varphi_{\varepsilon_0}+\varphi_1(\omega,1,0)}+\omega^{\varphi_{\varepsilon_0}+\varphi_1}}$
$\varphi_{\varepsilon_0}+\varphi_1(\omega,1,0)+\varphi_1$	$\varphi_{\varepsilon_0}+\varphi_1(\omega,1,0)+\varphi_1$	$\omega^{\omega^{(\varphi_{\varepsilon_0}2)}+\omega^{\varphi_{\varepsilon_0}+\varphi_1(\omega,1,0)}+\omega^{\varphi_{\varepsilon_0}+\varphi_1}}$

Table 19: `IterFuncOrdinal` exponential examples

8 Countable admissible ordinals

The first admissible ordinal is ω. ω_1^{CK} (the Church-Kleene ordinal) is the second. This latter is the ordinal of the recursive ordinals. For simplicity it will be written as ω_1 (see note 5). Gerald Sacks proved that the countable admissible ordinals are those defined like ω_1, but using Turing Machines with oracles (see Note 9)[14]. For example ω_2 is the set of all ordinals whose structure can be enumerated by a TM with an oracle that defines the structure of all recursive ordinals.

The completed infinite totality of a Turing Machine oracle can be avoided by generalizing the concept of a well founded recursive process. A first order well founded recursive process is one that accepts an indefinite number of integer inputs and halts for every possible sequence of these inputs. Recursive ordinals are the well orderings definable by such a process. This definition can be iterated by defining a process of type $x + 1$ to be well founded for all sequences of processes of type x. (Type 0 is the integers.) This can be iterated up to any countable ordinal. The admissible ordinals are a very sparse set of limit ordinals. In this document admissible *level* ordinals are all ordinals $\geq \omega_1^{CK}$, the ordinal of the recursive ordinals. The 'admissible index' refers to the κ index in ω_κ.

8.1 Typed parameters and `limitOrd`

Notations defined here give the structure of ordinals through the `limitElement` and `compare` functions. For admissible level ordinals `compare` is adequate, although it is now operating on an incomplete domain. However `limitElement` can no longer define how a limit ordinal is built up from smaller ordinals, because admissible level ordinal notations cannot always be defined as the limit of a recursive sequence of smaller notations.

As an adjunct to `Ordinal::limitElement`, `Ordinal::limitOrd` is defined. It accepts ordinal notations α less than ω_β where β is the `limitType` of the ordinal `limitOrd` is called from. The value of `limitType` is accessed with a member function of that name.

For example one can can define `limitOrd` for ω_1 to be the identity function that accepts any notation for a recursive ordinal as input and outputs its input. Then the union of the outputs for inputs satisfying this criteria is ω_1, This is not the way `limitOrd` is defined. It is used to define new ordinals that help to fill the gaps between admissible levels, but this illustrates how `limitOrd` is able to partially provide the defining function that `limitElement` serves for recursive ordinal notations. `limitElement` is restricted to integer inputs. In contrast `limitOrd` has a complex set of rules to specify which parameters are legal in a given context.

Table 20 shows the maximum value of `limitType` over ranges of ordinal values. Note whenever the admissible index (α in w_α) is a limit ordinal, the `limitType` of ω_α is defined to be the `limitType` of α. `limitType` is fully defined in Section 8.3. An additional function, α.`maxLimitType`(), returns the maximum value of `limitType` for any ordinal $\beta \leq \alpha$. α.`limitOrd`(β) is defined iff β.`maxLimitType`() $<$ α.`limitType`()

The ordinal represented by a notation α is the union of α.`limitOrd`(β) for all β such that β.`maxLimitType`() $<$ α.`limitType`()[38] In contrast to recursive ordinals, the notations

[38]The theory of admissible level ordinals is in some ways similar to the cardinals in set theory. In set

α	Maximum α.`limitType`() in range
0 or a successor ($< \omega_1$)	0 (`nullLimitType`)
recursive limit ordinal ($< \omega_1$)	1 (`integerLimitType`)
$\omega_1 \leq \alpha < \omega_2$	2 (`recOrdLimitType`)
$\omega_n \leq \ \alpha < \omega_{n+1}$	n + 1
$\alpha = \omega_\omega$	1 (`integerLimitType`)
$\omega_{\omega+5} \leq \alpha < \omega_{\omega+6}$	$\omega + 5$
$\alpha = \omega_{\omega_1}$	2 (`recOrdLimitType`)
$\omega_{\omega_1+1} \leq \alpha < \omega_{\omega_1+2}$	$\omega_1 + 1$

Table 20: Value of `limitType`

whose union is an admissible level ordinal ($\geq \omega_1$) are not recursively enumerable. However the operation that goes from some notation `not1` to `limitOrd(not1)` is recursive for the domain of notations defined at any point in time.

The explicitly typed hierarchy in this document is a bit reminiscent of the explicitly typed hierarchy in Whitehead and Russell's *Principia Mathematica*[16].

8.2 Admissible level ordinal notations

Notation systems are expanded to represent larger ordinals than those represented already. At the admissible level they can be expanded to partially fill the gaps in a necessarily incomplete system. ω_α.`limitOrd`(β) is defined as $\omega_\alpha[\beta]$. The idea is that this new parameter starts by diagonalizing what is definable by previous defined notations. This begins with ω_1.`limitOrd`(1) or $\omega_1[1]$. This is defined to be the limit of the recursive ordinals definable in the `IterFuncOrdinal class`. This is the sequence $\omega, \varphi_\omega, \varphi_{\varphi_\omega+1}, \varphi_{\varphi_{\varphi_\omega+1}+1}, ...$ The union of that sequence is represented by $\omega_1[1]$. For an ordinal, β, $\omega_1[\beta+1]$ is defined as the union of the sequence

$$\omega_1[\beta], \varphi_{\omega_1[\beta]+1}, \varphi_{\varphi_{\omega_1[\beta]+1}+1}, \varphi_{\varphi_{\varphi_{\omega_1[\beta]+1}+1}+1}, ..., .$$

For β a limit, $\omega_\alpha[\beta]$ is the union of $\omega_\alpha[\gamma]$ for $\gamma < \beta$. A complete description of the `limitElement` and `limitOrd` functions is given in Sections 9.2 and 9.5.

The normal form notation for one term of an admissible level ordinal adds a parameter of the admissible index (the most significant parameter) and a parameter in square brackets which signifies a *smaller* ordinal then the same term without an appended square bracketed parameter. There is a third option using double square brackets, [[]] to facilitate a form or collapsing described in Section 8.4.

The rest of the equation for one term in the normal form is the same as that for an `IterFuncOrdinal` shown in Equation 7. The C++ `class` for a single term of an admissible level ordinal notation is `AdmisNormalElement` and the `class` for an admissible level ordinal notation is `AdmisLevOrdinal`.

The `AdmisNormalElement` term of an `AdmisLevOrdinal` is one of the following forms.

theory there is a cardinal that is the union of finite iterations of the power set axiom. It is the union of a countable number of smaller sets. This contrasts with set of all reals which is not constructable from a smaller collection of smaller sets. Such sets are called regular cardinalss. (This definition requires the .)

Ordinal	**LimitElements**		
	1	2	3
$[[1]]\omega_1$	$\omega_1[\omega]$	$\omega_1[\omega_1[\omega]]$	$\omega_1[\omega_1[\omega_1[\omega]]]$
$[[1]]\omega_2$	$[[1]]\omega_1$	$[[1]]\omega_{1,[[1]]\omega_1+1}$	$[[1]]\omega_{1,[[1]]\omega_{1,[[1]]\omega_1+1}+1}$
$[[1]]\omega_3$	$[[1]]\omega_2$	$[[1]]\omega_{2,[[1]]\omega_2+1}$	$[[1]]\omega_{2,[[1]]\omega_{2,[[1]]\omega_2+1}+1}$
$[[2]]\omega_2$	$\omega_2[1]$	$\omega_2[2]$	$\omega_2[3]$
$[[2]]\omega_3$	$[[2]]\omega_3[1]$	$[[2]]\omega_3[2]$	$[[2]]\omega_3[3]$
$[[1]]\omega_\omega$	$[[1]]\omega_2$	$[[1]]\omega_3$	$[[1]]\omega_4$
$[[2]]\omega_\omega$	$[[2]]\omega_3$	$[[2]]\omega_4$	$[[2]]\omega_5$
$[[\omega]]\omega_\omega$	$[[1]]\omega_1$	$[[2]]\omega_2$	$[[3]]\omega_3$
$[[\omega^\omega]]\omega_{\omega^\omega}$	$[[\omega]]\omega_\omega$	$[[\omega^2]]\omega_{\omega^2}$	$[[\omega^3]]\omega_{\omega^3}$

Table 21: δ parameter examples

$$\omega_{\kappa,\gamma}(\beta_1, \beta_2, ..., \beta_m) \tag{8}$$

$$\omega_\kappa[\eta] \tag{9}$$

$$[[\delta]]\omega_\kappa[\eta] \tag{10}$$

$$[[\delta]]\omega_{\kappa,\gamma}(\beta_1, \beta_2, ..., \beta_m) \tag{11}$$

The γ and β_i parameters are defined as they are in Equation 7. The existence of any of these parameters defines a larger ordinal then the same notation without them. In contrast to every parameter defined so far, the η parameter drops down to a lower level ordinal. For example, $\omega_1[\alpha]$ is a recursive ordinal necessarily smaller than ω_1. The η parameter is only defined when γ and β_i are zero and κ is a successor.

The δ parameter, like the η parameter, defines a smaller ordinal than the same expression without this parameter. In particular it “collapses” the expression down to an ordinal $< \omega_\delta$. See Section 8.4 for a general description of collapsing. Table 21 gives some examples of how the δ parameter is defined.

The idea of the δ parameter is to collapse an arbitrarily large subset of the entire structure of countable ordinal notations *as they are defined by a finite formal system at a specific stage in its development*[39] at various points in the hierarchy. This is a bit like the Mandelbrot set[10] that embeds its entire structure within itself in an expanding infinite tree.

8.3 `limitType` of admissible level notations

The κ parameter in equations 8 to 11 determines the `limitType` of an admissible level notation with no other nonzero parameters. If κ is a limit then ω_κ.`limitType`() = κ.`limitType`(). If κ is a finite successor then ω_κ.`limitType`() $= \kappa + 1$. If $\kappa > \omega$ and κ is a successor then ω_κ.`limitType`() $= \kappa$.

[39]The underlying philosophy is that a finite formal system is an intrinsically incomplete structure that can always be expanded.

If an admissible level ordinal, α, has parameters other than κ, the least significant parameter determines the `limitType` of α. If the least significant parameter, β, is a limit ordinal then the `limitType` of β is the the `limitType` of α. if the least significant parameter of the notation is a successor then the `limitType` of the notation is 1 (or integer). In this case the limit is an infinite sequence of nested expressions. These rules are summarized in Table 22.

The δ parameter in equation 11 modifies the definition of `limitType`. If all parameters except δ and κ are 0 and δ is a successor, then δ determines the `limitType` just as κ did in the above paragraph. A nonzero δ parameter puts an upper bound on the `limitType` of any ordinal no matter what other parameters it has. The complete rules for `limitType` are in Section 9.3 and those for `maxLimitType` are in Section 9.4.

8.4 Ordinal collapsing

Collapsing[40]expands recursive ordinal notations using ordinals of higher type (admissible level or uncountable) to expand the hierarchy of recursive ordinal notations[1, 12]. There is an element of collapsing with the η parameter in square brackets defined in Section 8.2 in that it diagonalizes the earlier definitions of recursive ordinals by referencing a higher level notation. Before describing collapsing with the δ parameter, we give a brief overview of an existing approach.

One can define a collapsing function, $\Psi(\alpha)$ on ordinals to countable ordinals[41]. $\Psi(\alpha)$ is defined using a function $C(\alpha)$, from ordinals to sets of ordinals. $C(\alpha)$ is defined inductively on the integers for each ordinal α using $\Psi(\beta)$ for $\beta < \alpha$.

- $C(\alpha)_0 = \{0, 1, \omega, \Omega\}$ (Ω, is the ordinal of the countable ordinals.)
- $C(\alpha)_{n+1} = C(\alpha)_n \cup \{\beta_1 + \beta_2, \beta_1\beta_2, {\beta_1}^{\beta_2} : \beta_1, \beta_2 \in C(\alpha)_n\} \cup \{\Psi(\beta) : \beta \in C(\alpha)_n \wedge \beta < \alpha\}$.
- $C(\alpha) = \bigcup_{n\in\omega} C(\alpha)_n$.
- $\Psi(\alpha)$ is defined as the *least* ordinal not in $C(\alpha)$.

$\Psi(0) = \varepsilon_0$ because ε_0 is the least ordinal not in $C(0)$. (ε_0 is the limit of $\omega, \omega^{\omega}\omega^{\omega^{\omega}}, ...,$.) Similarly $\Psi(1) = \varepsilon_1$, because $C(1)$ includes $\Psi(0)$ which is ε_0. For a while $\Psi(\alpha) = \varepsilon_\alpha$. This stops at $\varphi(2, 0)$ which is the first fixed point of $\alpha \mapsto \varepsilon_\alpha$. $\Psi(\varphi(2, 0)) = \varphi(2, 0)$ but $\Psi(\varphi(2, 0) + 1) = \varphi(2, 0)$ also. The function remains static because $\varphi(2, 0)$ is not in $C(\alpha)$ for $\alpha \leq \Omega$.

$\Psi(\Omega) = \varphi(2, 0)$, but Ω was defined to be in $C(\alpha)_0$ and thus $C(\Omega + 1)$ includes $\Psi(\Omega)$ which is $\varphi(2, 0)$. Thus $\Psi(\Omega + 1) = \varepsilon_{\varphi(2,0)+1}$. $\Psi(\alpha)$ becomes static again at $\alpha = \varphi(2, 1)$ the second fixed point of $\alpha \mapsto \varepsilon_\alpha$. However ordinals computed using Ω support getting past

[40]Ordinal collapsing is also known as projection. I prefer the term collapsing, because I think the proofs have more to do with syntactical constructions of mathematical language than they do with an abstract projection in a domain the symbols refer to. Specifically I think cardinal numbers, on which projection is most often based, have only a relative meaning. See Section 10 for more about this philosophical approach

[41]This description is largely based on the Wikipedia article on "Ordinal collapsing function" as last modified on April 14, 2009 at 15:48. The notation is internally consistent in this document and differs slightly from the Wikipedia notation.

Rules		
	Case	`limitType`
1	$\alpha = \omega_\kappa$ κ a successor	κ is finite $\rightarrow \kappa + 1$ else κ
2	least significant parameter, α, is limit	α.`limitType`()
3	γ a successor and least significant	`integerLimitType = 1`
4	2 least significant parameters are successors	`integerLimitType = 1`
5	least significant parameter is successor and next least, α, is a limit	α`.limitType()`

Examples		
α	α.`limitType`()	Rule
ω_1	2	1
ω_2	3	1
ω_{2,ω^ω}	1	2
ω_ω	1	2
$\omega_{\omega_{\omega^\omega},4}$	1	2
$\omega_{\omega,5}(3)$	1	4
$\omega_{\omega,\omega_1}(4)$	2	5
$\omega_{\omega_{\omega^\omega+1,5},5}$	1	3
$\omega_{\omega_4}(3)$	1	5
$\omega_{\omega+5}$	$\omega + 5$	1
$\omega_{\omega,\omega_{\omega+5}}$	$\omega + 5$	2
$\omega_{\omega_{\omega^{\omega^{(\omega_1 2)}}+6}}(3, 5, \omega_{\omega^{\omega^{(\omega_1 2)}}+6})$	$\omega^{\omega^{(\omega_1 2)}} + 6$	2
$\omega_{\omega_5,3}$	1	3
$\omega_{\omega_{\omega^{\omega^{(\omega_1 2)}}+6}}$	$\omega^{\omega^{(\omega_1 2)}} + 6$	2
$\omega_{\omega,\omega_{\omega_{\omega^{\omega^{(\omega_1 2)}}+6}}}$	$\omega^{\omega^{(\omega_1 2)}} + 6$	2
$\omega_{\omega,\omega_1+6}$	1	5

Table 22: `AdmisNormalElement::limitTypes`

fixed points in the same way that Ω did. The first case like this is $\Psi(\Omega 2) = \varphi(2,1)$ and thus $\Psi(\Omega 2+1) = \varepsilon_{\varphi(2,1)+1}$.

Each addition of 1 advances to the next ε value until Ψ gets stuck at a fixed point. Each addition of Ω moves to the next fixed point of $\alpha \mapsto \varepsilon_\alpha$ so that $\Psi(\Omega(1+\alpha)) = \varphi(2,\alpha)$ for $\alpha < \varphi(3,0)$. Powers of Ω move further up the Veblen hierarchy: $\Psi(\Omega^2) = \varphi(3,0)$, $\Psi(\Omega^\beta) = \varphi(1+\beta,0)$ and $\Psi(\Omega^\beta(1+\alpha)) = \varphi(1+\beta,\alpha)$. Going further $\Psi(\Omega^\Omega) = \Gamma(0) = \varphi(1,0,0)$, $\Psi(\Omega^{\Omega(1+\alpha)}) = \varphi(\alpha,0,0)$ and $\Psi(\Omega^{\Omega^2}) = \varphi(1,0,0,0)$,

Collapsing, as defined here, connects basic ordinal arithmetic (addition, multiplication and exponentiation) to higher level ordinal functions. Ordinal arithmetic on Ω gets past fixed points in the definition of Ψ until we reach an ordinal that is not in $C(\alpha)$ either by definition or by basic ordinal arithmetic on ordinals in $C(\alpha)$. This is the ordinal $\varepsilon_{\Omega+1}$[42]. $\Psi(\varepsilon_{\Omega+1})= \Psi(\Omega) \bigcup \Psi(\Omega^\Omega) \bigcup \Psi(\Omega^{\Omega^\Omega}) \bigcup \Psi(\Omega^{\Omega^{\Omega^\Omega}}) \bigcup ...$ This is the Bachmann-Howard ordinal[8]. It is the largest ordinal in the range of Ψ as defined above. $\Psi(\alpha)$ is a constant for $\alpha \geq \varepsilon_{\Omega+1}$ because there is no way to incorporate $\Psi(\varepsilon_{\Omega+1})$ into $C(\alpha)$.

[42]Note $\varepsilon_\Omega = \Omega$.

	Notation		
#	Ψ	$\varphi \quad \omega$	**Bound**
1	$\Psi(\alpha)$	ε_α	$\alpha < \varphi(2,0)$
2	$\Psi(\Omega 13)$	$\varphi(2,12)$	
3	$\Psi(\Omega(1+\alpha))$	$\varphi(2,\alpha)$	$\alpha < \varphi(3,0)$
4	$\Psi(\Omega^2 6)$	$\varphi(3,5)$	
5	$\Psi(\Omega^2(1+\alpha))$	$\varphi(3,\alpha)$	$\alpha < \varphi(4,0)$
6	$\Psi(\Omega^{11} 6)$	$\varphi(12,5)$	
7	$\Psi(\Omega^\beta(1+\alpha))$	$\varphi(1+\beta,\alpha)$	$\alpha < \varphi(1+\beta,0) \wedge \beta < \varphi(1,0,0)$
8	$\Psi(\Omega^\Omega)$	$\varphi(1,0,0) = \Gamma_0$	
9	$\Psi(\Omega^{\Omega 2})$	$\varphi(2,0,0)$	
10	$\Psi(\Omega^{(\Omega 2+3)} 6)$	$\varphi(2,3,5)$	
11	$\Psi(\Omega^{\Omega^2})$	$\varphi(1,0,0,0)$	
12	$\Psi(\Omega^{\Omega^n})$	$\varphi(1_1,0_2,...,0_{n+2})$	
13	$\Psi(\Omega^{\Omega^n \alpha_1})$	$\varphi(\alpha_1,0_2,...,0_{n+2})$	$\alpha_1 < \varphi(1_1,0_2,...,0_{n+3})$
14	$\Psi(\Omega^{\Omega^\omega})$	φ_1	
15	$\Psi(\Omega^{\Omega^\omega 5})$	$\varphi_1(5)$	
16	$\Psi(\Omega^{\Omega^\omega(1+\alpha)})$	$\varphi_1(\alpha)$	$\alpha < \varphi_1(1,0)$
17	$\Psi(\Omega^{(\Omega^\omega(\Omega 4+2))})$	$\varphi_1(4,2)$	
18	$\Psi(\Omega^{\Omega^{\omega 2}})$	φ_2	
19	$\Psi(\Omega^{\Omega^{\omega^2}})$	φ_ω	
20	$\Psi(\Omega^{\Omega^{\omega^\alpha}})$	φ_α	$\alpha < \omega_1[1]$
21	$\Psi(\Omega^{\Omega^\Omega})$	$\omega_1[1]$	
22	$\Psi(\Omega^{\Omega^{\Omega^\Omega}})$	$[[1]]\omega_1$	
23	$\Psi(\Omega^{\Omega^{\Omega^{\Omega^\Omega}}})$	$[[1]]\omega_2$	
24	$\Psi(\varepsilon_{\Omega+1})$	$[[1]]\omega_\omega$	

Table 23: Ψ collapsing function with bounds

	Notation	`limitElement`			
#	Ψ / φ ω	**1**	**2**	**3**	**4**
1	$\Psi(\Omega)$ $\varphi(2,0)$	$\Psi(1)$ ε_1	$\Psi(\Psi(1)+1)$ $\varepsilon_{\varepsilon_1+1}$	$\Psi(\Psi(\Psi(1)+1)+1)$ $\varepsilon_{\varepsilon_{\varepsilon_1+1}+1}$	$\Psi(\Psi(\Psi(\Psi(1)+1)+1)+1)$ $\varepsilon_{\varepsilon_{\varepsilon_{\varepsilon_1+1}+1}+1}$
2	$\Psi(\Omega^2)$ $\varphi(3,0)$	$\Psi(\Omega 2)$ $\varphi(2,1)$	$\Psi(\Omega(\Psi(\Omega 2)+1))$ $\varphi(2,\varphi(2,1)+1)$	$\Psi(\Omega(\Psi(\Omega(\Psi(\Omega 2)+1))+1))$ $\varphi(2,\varphi(2,\varphi(2,1)+1)+1)$	$\Psi(\Omega(\Psi(\Omega(\Psi(\Omega(\Psi(\Omega 2)+1))+1))+1))$ $\varphi(2,\varphi(2,\varphi(2,\varphi(2,1)+1)+1)+1)$
3	$\Psi(\Omega^\omega)$ $\varphi(\omega,0)$	$\Psi(0)$ ε_0	$\Psi(\Omega)$ $\varphi(2,0)$	$\Psi(\Omega^2)$ $\varphi(3,0)$	$\Psi(\Omega^3)$ $\varphi(4,0)$
4	$\Psi(\Omega^\Omega)$ Γ_0	$\Psi(0)$ ε_0	$\Psi(\Omega^{\Psi(0)+1})$ $\varphi(\varepsilon_0+1,0)$	$\Psi(\Omega^{\Psi(\Omega^{\Psi(0)+1})+1})$ $\varphi(\varphi(\varepsilon_0+1,0)+1,0)$	$\Psi(\Omega^{\Psi(\Omega^{\Psi(\Omega^{\Psi(0)+1})+1})+1})$ $\varphi(\varphi(\varphi(\varepsilon_0+1,0)+1,0)+1,0)$
5	$\Psi(\Omega^{\Omega^2})$ $\varphi(1,0,0,0)$	$\Psi(\Omega^\Omega)$ Γ_0	$\Psi(\Omega^{\Omega\Psi(\Omega^\Omega)+1})$ $\varphi(\Gamma_0+1,0,0)$	$\Psi(\Omega^{\Omega\Psi(\Omega^{\Omega\Psi(\Omega^\Omega)+1})+1})$ $\varphi(\varphi(\Gamma_0+1,0,0)+1,0,0)$	$\Psi(\Omega^{\Omega\Psi(\Omega^{\Omega\Psi(\Omega^{\Omega\Psi(\Omega^\Omega)+1})+1})+1})$ $\varphi(\varphi(\varphi(\Gamma_0+1,0,0)+1,0,0)+1,0,0)$
6	$\Psi(\Omega^{\Omega^\omega})$ φ_1	ω ω	$\Psi(0)$ ε_0	$\Psi(\Omega^\Omega)$ Γ_0	$\Psi(\Omega^{\Omega^2})$ $\varphi(1,0,0,0)$
7	$\Psi(\Omega^{\Omega^{\omega 2}})$ φ_2	$\Psi(\Omega^{\Omega^\omega\Psi(\Omega^{\Omega^\omega})+1})$ $\varphi_1(\varphi_1+1)$	$\Psi(\Omega^{(\Omega^\omega(\Omega\Psi(\Omega^{\Omega^\omega})+1))})$ $\varphi_1(\varphi_1+1,0)$	$\Psi(\Omega^{(\Omega^\omega(\Omega^2\Psi(\Omega^{\Omega^\omega})+1))})$ $\varphi_1(\varphi_1+1,0,0)$	$\Psi(\Omega^{(\Omega^\omega(\Omega^3\Psi(\Omega^{\Omega^\omega})+1))})$ $\varphi_1(\varphi_1+1,0,0,0)$
8	$\Psi(\Omega^{\Omega^{\omega^2}})$ φ_ω	$\Psi(\Omega^{\Omega^\omega})$ φ_1	$\Psi(\Omega^{\Omega^{\omega 2}})$ φ_2	$\Psi(\Omega^{\Omega^{\omega 3}})$ φ_3	$\Psi(\Omega^{\Omega^{\omega 4}})$ φ_4
9	$\Psi(\Omega^{\Omega^\Omega})$ $\omega_1[1]$	ω ω	$\Psi(\Omega^{\omega^2})$ φ_ω	$\Psi(\Omega^{\Omega^{\Psi(\Omega^{\Omega^{\omega^2}})+\omega}})$ $\varphi_{\varphi_\omega+1}$	$\Psi(\Omega^{\Omega^{\Psi(\Omega^{\Omega^{\Psi(\Omega^{\Omega^{\omega^2}})+\omega}})+\omega}})$ $\varphi_{\varphi_{\varphi_\omega+1}+1}$
10	$\Psi(\Omega^{\Omega^{\Omega\omega}})$ $\omega_1[\omega]$	$\Psi(\Omega^{\Omega^\Omega})$ $\omega_1[1]$	$\Psi(\Omega^{\Omega^{\Omega 2}})$ $\omega_1[2]$	$\Psi(\Omega^{\Omega^{\Omega 3}})$ $\omega_1[3]$	$\Psi(\Omega^{\Omega^{\Omega 4}})$ $\omega_1[4]$
11	$\Psi(\Omega^{\Omega^{\Omega^\Omega}})$ $[[1]]\omega_1$	$\Psi(\Omega^{\Omega^{\Omega\omega}})$ $\omega_1[\omega]$	$\Psi(\Omega^{\Omega^{\Omega\Psi(\Omega^{\Omega^{\Omega\omega}})}})$ $\omega_1[\omega_1[\omega]]$	$\Psi(\Omega^{\Omega^{\Omega\Psi(\Omega^{\Omega^{\Omega\Psi(\Omega^{\Omega^{\Omega\omega}})}})}})$ $\omega_1[\omega_1[\omega_1[\omega]]]$	$\Psi(\Omega^{\Omega^{\Omega\Psi(\Omega^{\Omega^{\Omega\Psi(\Omega^{\Omega^{\Omega\Psi(\Omega^{\Omega^{\Omega\omega}})}})}})}})$ $\omega_1[\omega_1[\omega_1[\omega_1[\omega]]]]$
12	$\Psi(\Omega^{\Omega^{\Omega^{\Omega^\Omega}}})$ $[[1]]\omega_2$	$\Psi(\Omega^{\Omega^{\Omega^\Omega}})$ $[[1]]\omega_1$	$\Psi(\Omega^{\Omega^{\Omega(\Omega\Omega^{\Psi(\Omega^{\Omega^{\Omega^\Omega}})+\omega})}})$ $[[1]]\omega_{1,[[1]]\omega_1+1}$	$\Psi(\Omega^{\Omega^{\Omega(\Omega\Omega^{\Psi(\Omega^{\Omega^{\Omega(\Omega\Omega^{\Psi(\Omega^{\Omega^{\Omega^\Omega}})+\omega})}})+\omega})}})$ $[[1]]\omega_{1,[[1]]\omega_{1,[[1]]\omega_1+1}+1}$	$\Psi(\Omega^{\Omega^{\Omega(\Omega\Omega^{\Psi(\Omega^{\Omega^{\Omega(\Omega\Omega^{\Psi(\Omega^{\Omega^{\Omega(\Omega\Omega^{\Psi(\Omega^{\Omega^{\Omega^\Omega}})+\omega})}})+\omega})}})+\omega})}})$ $[[1]]\omega_{1,[[1]]\omega_{1,[[1]]\omega_{1,[[1]]\omega_1+1}+1}+1}$
13	$\Psi(\varepsilon_{\Omega+1})$ $[[1]]\omega_\omega$	$\Psi(\Omega^{\Omega^{\Omega^{\Omega^\Omega}}})$ $[[1]]\omega_2$	$\Psi(\Omega^{\Omega^{\Omega^{\Omega^{\Omega^\Omega}}}})$ $[[1]]\omega_3$	$\Psi(\Omega^{\Omega^{\Omega^{\Omega^{\Omega^{\Omega^\Omega}}}}})$ $[[1]]\omega_4$	$\Psi(\Omega^{\Omega^{\Omega^{\Omega^{\Omega^{\Omega^{\Omega^\Omega}}}}}})$ $[[1]]\omega_5$

Table 24: Ψ collapsing function at critical limits

Table 23 connects the notations defined by the Ψ function to those defined with equations 8 to 11. Lines up to 12 follow from the above description. This line uses parameter subscripts to indicate the number of parameters. Otherwise it is straightforward. In line 14 φ_1 is the first fixed point not reachable from Veblen functions with a finite number of parameters. $\varphi_1 = \bigcup_{n\in\omega} \varphi(1_1, 0_2, 0_3 ..., 0_n)$. Lines 18, 19 and 20 illustrate how each increment of α by one in φ_α adds a factor of ω to the the highest order exponent in Ψ notation. This maps to the definition of φ_α in Section 5.5.

Table 24 provides additional detail to relate the Ψ notation to the notation in this document. It covers the range of the Ψ function at critical limits. For each entry the table gives the ordinal notation and first 4 elements in a limit sequence that converges to this ordinal. This same information is repeated in paired lines. The first is in Ψ notation and the second in the notation defined in this document. It illustrates the conditions in which the Ψ notation requires another level of exponentiation. This happens with limiting sequences that involve ever higher levels of exponentiation of the largest ordinals defined at that level in the Ψ notation. This occurs in lines 4, 9, 11 and 12 of the table corresponding to Ω raised to the second through fifth powers.

8.5 Displaying `Ordinals` in Ψ format

Function `Ordinal::psiNormalForm` provides a display option for the Ψ function format in addition to the display functions described in Section 4.1. It provides a limited display capability. Not all values can be converted. This is due in part to the erratic nature of Ψ as it gets stuck at various fixed points. The purpose is to provide an automated conversion that handles the primary cases throughout the Ψ hierarchy just defined. If a value is not displayable then the string '`not` Ψ `displayable`' is output in TEX format. Many of the tables in this section use `psiNormalForm`. It is accessible in the interactive calculator as described in Section A.8.2 under the `opts` command.

8.6 Admissible level ordinal collapsing

For any two ordinals ω_α and $\omega_{\alpha+1}$ there is an unclosable gap into which one can, in a sense, fit any finite notation system that has been or ever will be developed. The only well orderings fully definable in a finitely axiomatizable system are, by definition, recursively enumerable. As mentioned before finite formulations of a fragment of the countable admissible level hierarchy are a bit like the Mandelbrot set where the entire structure is embedded again and again at many points in an infinite tree[43]

The idea is to "freeze" the structure at some point in its development and then embed this frozen image in an unfrozen version of itself. It is a bit like taking recursion to a higher level of abstraction. This is accomplished with notations in the form of equations 10 ($[[\delta]]\omega_\kappa[\eta]$) and 11 ($[[\delta]]\omega_{\kappa,\gamma}(\beta_1, \beta_2, ..., \beta_m)$). They both define a value $< \omega_\delta$. It is required that $\kappa \geq \delta$[44]. The idea is to use notations up through any ω_κ definable in the frozen system to define ordinal notations $< \omega_\delta$ in the unfrozen system.

[43]Ordinal hierarchies have infinite ascending chains (for example the positive integers) but cannot have infinite descending chains (like the negative integers).

[44]Note that any notation from equations 10 and 11 that starts with [[1]] must define a recursive ordinal.

This starts by diagonalizing what can be defined with $\omega_\kappa[\eta]$. See Table 25 for the definition of $[[1]]\omega_1$. The complete definition of δ (and the other parameters in equations 10 and 11) are in sections 9.2 and 9.5 that document the `limitElement` and `limitOrd` member functions.

8.7 Functional hierarchies and collapsing

In sections 2.2 and 8 ordinals were described in terms of well founded functional hierarchies. The idea is that one starts with processes well founded for arbitrary sequences of integers. Such processes are called type 1. Next consider processes well founded for arbitrary sequences of Gödel numbers of type 1 processes. This can be iterated up to any integer and iterated further up to any ordinal definable by this process. At limit ordinals one considers a process that is well founded for all lower type processes.

In constructing a functional hierarchy like this one makes it explicitly incomplete and general enough to be expandable to any level in the hierarchies outlined above. One can think of collapsing in this context as dropping down the hierarchy to lower levels using specific parameter sequences.

α	$\alpha.\texttt{limitElement}(n)$			
n	**1**	**2**	**3**	**4**
$[[1]]\omega_1$	$\omega_1[\omega]$	$\omega_1[\omega_1[\omega]]$	$\omega_1[\omega_1[\omega_1[\omega]]]$	$\omega_1[\omega_1[\omega_1[\omega_1[\omega]]]]$
$\omega_1[\omega]$	$\omega_1[1]$	$\omega_1[2]$	$\omega_1[3]$	$\omega_1[4]$
$\omega_1[\omega_1[\omega]]$	$\omega_1[\omega_1[1]]$	$\omega_1[\omega_1[2]]$	$\omega_1[\omega_1[3]]$	$\omega_1[\omega_1[4]]$
$\omega_1[\omega_1[\omega_1[\omega]]]$	$\omega_1[\omega_1[\omega_1[1]]]$	$\omega_1[\omega_1[\omega_1[2]]]$	$\omega_1[\omega_1[\omega_1[3]]]$	$\omega_1[\omega_1[\omega_1[4]]]$
$\omega_1[\omega_1[\omega_1[\omega_1[\omega]]]]$	$\omega_1[\omega_1[\omega_1[\omega_1[1]]]]$	$\omega_1[\omega_1[\omega_1[\omega_1[2]]]]$	$\omega_1[\omega_1[\omega_1[\omega_1[3]]]]$	$\omega_1[\omega_1[\omega_1[\omega_1[4]]]]$
$\omega_1[1]$	ω	φ_{ω}	$\varphi_{\varphi_{\omega}+1}$	$\varphi_{\varphi_{\varphi_{\omega}+1}+1}$
$\omega_1[2]$	$\omega_1[1]$	$\varphi_{\omega_1[1]+1}$	$\varphi_{\varphi_{\omega_1[1]+1}+1}$	$\varphi_{\varphi_{\varphi_{\omega_1[1]+1}+1}+1}$
$\omega_1[3]$	$\omega_1[2]$	$\varphi_{\omega_1[2]+1}$	$\varphi_{\varphi_{\omega_1[2]+1}+1}$	$\varphi_{\varphi_{\varphi_{\omega_1[2]+1}+1}+1}$
$\omega_1[4]$	$\omega_1[3]$	$\varphi_{\omega_1[3]+1}$	$\varphi_{\varphi_{\omega_1[3]+1}+1}$	$\varphi_{\varphi_{\varphi_{\omega_1[3]+1}+1}+1}$
$\omega_1[\omega_1[1]]$	$\omega_1[\omega]$	$\omega_1[\varphi_{\omega}]$	$\omega_1[\varphi_{\varphi_{\omega}+1}]$	$\omega_1[\varphi_{\varphi_{\varphi_{\omega}+1}+1}]$
$\omega_1[\omega_1[2]]$	$\omega_1[\omega_1[1]]$	$\omega_1[\varphi_{\omega_1[1]+1}]$	$\omega_1[\varphi_{\varphi_{\omega_1[1]+1}+1}]$	$\omega_1[\varphi_{\varphi_{\varphi_{\omega_1[1]+1}+1}+1}]$
$\omega_1[\omega_1[3]]$	$\omega_1[\omega_1[2]]$	$\omega_1[\varphi_{\omega_1[2]+1}]$	$\omega_1[\varphi_{\varphi_{\omega_1[2]+1}+1}]$	$\omega_1[\varphi_{\varphi_{\varphi_{\omega_1[2]+1}+1}+1}]$
$\omega_1[\omega_1[4]]$	$\omega_1[\omega_1[3]]$	$\omega_1[\varphi_{\omega_1[3]+1}]$	$\omega_1[\varphi_{\varphi_{\omega_1[3]+1}+1}]$	$\omega_1[\varphi_{\varphi_{\varphi_{\omega_1[3]+1}+1}+1}]$
$\omega_1[\omega_1[\omega_1[1]]]$	$\omega_1[\omega_1[\omega]]$	$\omega_1[\omega_1[\varphi_{\omega}]]$	$\omega_1[\omega_1[\varphi_{\varphi_{\omega}+1}]]$	$\omega_1[\omega_1[\varphi_{\varphi_{\varphi_{\omega}+1}+1}]]$
$\omega_1[\omega_1[\omega_1[2]]]$	$\omega_1[\omega_1[\omega_1[1]]]$	$\omega_1[\omega_1[\varphi_{\omega_1[1]+1}]]$	$\omega_1[\omega_1[\varphi_{\varphi_{\omega_1[1]+1}+1}]]$	$\omega_1[\omega_1[\varphi_{\varphi_{\varphi_{\omega_1[1]+1}+1}+1}]]$
$\omega_1[\omega_1[\omega_1[3]]]$	$\omega_1[\omega_1[\omega_1[2]]]$	$\omega_1[\omega_1[\varphi_{\omega_1[2]+1}]]$	$\omega_1[\omega_1[\varphi_{\varphi_{\omega_1[2]+1}+1}]]$	$\omega_1[\omega_1[\varphi_{\varphi_{\varphi_{\omega_1[2]+1}+1}+1}]]$
$\omega_1[\omega_1[\omega_1[4]]]$	$\omega_1[\omega_1[\omega_1[3]]]$	$\omega_1[\omega_1[\varphi_{\omega_1[3]+1}]]$	$\omega_1[\omega_1[\varphi_{\varphi_{\omega_1[3]+1}+1}]]$	$\omega_1[\omega_1[\varphi_{\varphi_{\varphi_{\omega_1[3]+1}+1}+1}]]$
$\omega_1[\omega_1[\omega_1[\omega_1[1]]]]$	$\omega_1[\omega_1[\omega_1[\omega]]]$	$\omega_1[\omega_1[\omega_1[\varphi_{\omega}]]]$	$\omega_1[\omega_1[\omega_1[\varphi_{\varphi_{\omega}+1}]]]$	$\omega_1[\omega_1[\omega_1[\varphi_{\varphi_{\varphi_{\omega}+1}+1}]]]$
$\omega_1[\omega_1[\omega_1[\omega_1[2]]]]$	$\omega_1[\omega_1[\omega_1[\omega_1[1]]]]$	$\omega_1[\omega_1[\omega_1[\varphi_{\omega_1[1]+1}]]]$	$\omega_1[\omega_1[\omega_1[\varphi_{\varphi_{\omega_1[1]+1}+1}]]]$	$\omega_1[\omega_1[\omega_1[\varphi_{\varphi_{\varphi_{\omega_1[1]+1}+1}+1}]]]$
$\omega_1[\omega_1[\omega_1[\omega_1[3]]]]$	$\omega_1[\omega_1[\omega_1[\omega_1[2]]]]$	$\omega_1[\omega_1[\omega_1[\varphi_{\omega_1[2]+1}]]]$	$\omega_1[\omega_1[\omega_1[\varphi_{\varphi_{\omega_1[2]+1}+1}]]]$	$\omega_1[\omega_1[\omega_1[\varphi_{\varphi_{\varphi_{\omega_1[2]+1}+1}+1}]]]$
$\omega_1[\omega_1[\omega_1[\omega_1[4]]]]$	$\omega_1[\omega_1[\omega_1[\omega_1[3]]]]$	$\omega_1[\omega_1[\omega_1[\varphi_{\omega_1[3]+1}]]]$	$\omega_1[\omega_1[\omega_1[\varphi_{\varphi_{\omega_1[3]+1}+1}]]]$	$\omega_1[\omega_1[\omega_1[\varphi_{\varphi_{\varphi_{\omega_1[3]+1}+1}+1}]]]$

Table 25: Structure of $[[1]]\omega_1$

9 AdmisLevOrdinal class

The `AdmisLevOrdinal class` goes beyond the recursive ordinals to construct notations for countable admissible ordinals beginning with the Church-Kleene or second admissible ordinal[45]. This is the ordinal of the recursive ordinals. `class AdmisLevOrdinal` is derived from base classes starting with `IterFuncOrdinal`. The `Ordinal`s it defines can be used in constructors for the `Ordinal` base `class`es it is derived from. However `AdmisLevOrdinal`s are different from ordinals constructed with those base classes. For every recursive ordinal there is a finite notation system that can enumerate a notation for that ordinal and every ordinal less than it such that there is a recursive ordering of the notations isomorphic to the ordering of the ordinals they represent. This is not true for the ordinal of the recursive ordinals or any larger ordinal.

Three new member functions: `limitOrd`, `limitType` and `maxLimitType` are needed for this class to deal with the limitations that exist at this level of the ordinal hierarchy. These functions are outlined in Section 8 and described in detail in sections 9.3 (for `limitType`), 9.4 (for `maxLimitType`) and 9.5 (for `limitOrd`).

`C++ class AdmisLevOrdinal` is derived from `class IterFuncOrdinal` which in turn is derived from `FiniteFuncOrdinal` and `Ordinal`. It implements an iterative function hierarchy syntactically similar to that described in Section 5.5. It uses an expanded version of the normal form in Equation 7 given in equations 8 to 11. The `class` for a single term of an `AdmisLevOrdinal` is `AdmisNormalElement`. It is derived from `IterFuncNormalElement` and its base classes.

All parameters (except κ) can be zero. Omitted parameters are set to 0. To omit κ use the notation for an `IterFuncOrdinal` in Equation 7. κ is the admissible index, i. e. (the κ in ω_κ). ω_1 is the ordinal of the recursive ordinals or the smallest ordinal that is not recursive. The expanded notation at this level is specified in Section 8.2. Parameters γ and β_i have a definition similar to that in Table 10 and are explained in section 9.2 on `AdmisNormalElement::limitElement` and section 9.5 on `AdmisNormalElement::limitOrd` as are the new parameters in this class: κ, η and δ.

The `AdmisLevOrdinal class` should not be used directly to create ordinal notations. Instead use function `admisLevelFunctional` which checks for fixed points and creates unique notations for each ordinal[46]. This function takes up to five arguments. The first two are required and the rest are optional. The first gives the admissible ordinal index, the κ in ω_κ. The second gives the level of iteration or the value of γ in equations 8 and 11. The third gives a `NULL` terminated array of pointers to `Ordinal`s which are the β_i in equations 8 and 11. This parameter is optional and can be created with function `createParameters` described in Section 6. The fourth parameter is an ordinal reference that defaults to 0. It is the value of η in Equations 9 and 10. The fifth parameter is an `Ordinal` reference which defaults to zero. It is the value of δ in equations 10 and 11. Some examples are shown in Table 26.

[45]The first admissible ordinal is the ordinal of the integers, ω.

[46] In the interactive ordinal calculator one can use notations like `[[delta]] omega_{ kappa, lambda} (b1,b2,...,bn)`. For more examples see sections A.11.7, A.11.8 and A.11.9.

'cp' stands for `createParameters` and 'af' stands for `admisLevelFunctional`

C++ code examples	**Ordinal**
`af(zero,zero,cp(&one))`	ω
`af(zero,one,cp(&one,&zero))`	$\varphi_1(1,0)$
`af(one,zero,cp(&one,&one,&zero))`	$\omega_1(1,1,0)$
`af(one,zero)`	ω_1
`af(one,zero,NULL,eps0)`	$\omega_1[\varepsilon_0]$
`af(one,zero,NULL,Ordinal::five)`	$\omega_1[5]$
`af(one,omega1CK,cp(&one))`	$\omega_{1,\omega_1}(1)$
`af(one,one,cp(&one,&omega1CK))`	$\omega_{1,1}(1,\omega_1)$
`af(one,Ordinal::two, cp(&one,&omega,&zero))`	$\omega_{1,2}(1,\omega,0)$
`af(omega,omega,cp(&one,&omega1CK,&zero,&zero))`	$\omega_{\omega,\omega}(1,\omega_1,0,0)$
`af(omega1CK,omega)`	$\omega_{\omega_1,\omega}$
`af(one,omega,cp(&iterativeFunctional(omega)))`	$\omega_{1,\omega}(\varphi_\omega)$
`af(af(one,zero)+1,zero,NULL,omega)`	$\omega_{\omega_1+1}[\omega]$
`af(Ordinal::two,zero,NULL,zero,Ordinal::two)`	$[[2]]\omega_2$
`af(Ordinal::omega,zero,NULL,zero,Ordinal::three)`	$[[3]]\omega_\omega$
`af(omega,zero,NULL,zero,omega)`	$[[\omega]]\omega_\omega$
`af(Ordinal::two,omega1CK,cp(&one),zero,Ordinal::two)`	$[[2]]\omega_{2,\omega_1}(1)$
`af(omega,omega,cp(&one,&omega1CK,&zero),zero,one)`	$[[1]]\omega_{\omega,\omega}(1,\omega_1,0)$

Table 26: `admisLevelFunctional` C++ code examples

9.1 `AdmisNormalElement::compare` member function

`AdmisNormalElement::compare` supports one term in the normal form in Equation 7 expanded with terms in the form of equations 8 to 11. `AdmisNormalElement::compare` compares its `CantorNormalElement` parameter `trm`, against the `AdmisNormalElement` instance `compare` is called from. As with `FiniteFuncOrdinals` and `IterFuncOrdinals`, the work of comparing `Ordinals` with multiple terms is left to the `Ordinal` and `OrdinalImpl` base class functions which call the `virtual` functions that operate on a single term.

`AdmisNormalElement::compare`, with a `CantorNormalElement` and an ignore factor flag as arguments, overrides `IterFuncNormalElement::compare` with the same argument (see Section 7.1). It outputs 1, 0 or -1 if the object it is called from is greater than equal to or less than its argument. There are two versions of `AdmisNormalElement::compare` the first has two arguments as described above. The second is for internal use only and has two additional 'context' arguments. the context for the base function and the context for the argument. The context is the value of the δ parameter in force at this stage of the compare. The three base classes on which `AdmisNormalElement` is built also support this context sensitive version of compare.

`compare` first checks if its argument's `codeLevel` is $>$ `admisCodeLevel`. If so it calls a higher level routine by calling its argument's member function. The code level of the `class` object that `AdmisNormalElement::compare` is called from should always be `admisCodeLevel`.

The δ parameter makes comparisons context sensitive. The δ values of the base object and `compare`'s parameter are relevant not just for comparing the two class instances but also for all comparisons of internal parameters of those instances. Thus context sensitive compare passes the δ values to `compare`s that are called recursively. For base classes that `AdmisNormalElement` is derived from, these additional arguments are only used to pass on to their internal `compare` calls. The δ values contexts only modify the meaning of ordinal notations for admissible level ordinals. The context sensitive version of the `virtual` function `compare` has four arguments.

- `const OrdinalImpl& embdIx` — the context for the base `class` object from which this compare originated.
- `const OrdinalImpl& termEmbdIx` — the context of the `CantorNormalElement` term at the start of the `compare`.
- `const CantorNormalElement& trm` — the term to be compared at this point in the comparison tree.
- `bool ignoreFactor` — an optional parameter that defaults to `false` and indicates that an integer `factor` is to be ignored in the comparison.

As with `compare` for `FiniteFuncOrdinal`s and `IterFuncOrdinal`s this function depends on the `getMaxParameter()` that returns the maximum value of all the parameters (except η and δ) in equations 8 and 11.

The logic of `AdmisNormalElement::compare` with the above four parameters is summarized in Table 28. The original version of `compare`, without the context arguments, calls the routine with context parameters.

9.2 `AdmisNormalElement::limitElement` member function

`AdmisNormalElement::limitElement` overrides `IterFuncNormalElement::limitElement` (see Section 6.2). It operates on a single term of the normal form expansion and does the bulk of the work, It takes a single integer parameter. Increasing values for the argument yield larger ordinal notations as output. In contrast to base `class` versions of this routine, in this version the union of the ordinals represented by the outputs for all integer inputs are not necessarily equal to the ordinal represented by the `AdmisNormalElement class` instance `limitElement` is called from. They are equal if the `limitType` of this instance of `AdmisNormalElement` is `integerLimitType` (see tables 20 and 38).

As usual `Ordinal::limitElement` does the work of operating on all but the last term of an `Ordinal` by copying all but the last term of the result unchanged from the input `Ordinal`. The last term is generated based on the last term of the `Ordinal` instance `limitElement` is called from. The three routines that implement `AdmisNormalElement::limitElement` are outlined in tables 30 to 32[47]. Some examples are shown in Tables 33 to 35.

[47]Routine `leUse` is used when κ is a limit and the least significant parameter. One needs to take an element from a sequence whose union is κ but one must insure that this does not lead to a value less than δ. One also must insure that increasing inputs produce increasing outputs and the output is always less than the input. The algorithm used has the same kernel code for this problem in `limitOrd`. See Note 48 on page 62 for a description of the algorithm.

Symbol	**Meaning**
`cmp`	Gnu Multiple Precision Arithmetic comparison routine
`compareFiniteParams`	compares β_i in equations 8 and 11 in object it is called from and its argument
δ_{ef}	effective (context dependent) value of δ
`diff`	temporary value with compare result
`effIndexCk`	minimum of κ, δ if $\delta > 0$ otherwise κ
`functionLevel`	γ parameter in equations 8 and 11
`ignoreFactor`	parameter to ignore factor in comparison
(ignf)	ignore factor and all but the first term
`maxParameter`	largest parameter of `class` instance or `trm`
`maxParamFirstTerm`	first term of largest parameter of `trm`
`isZ`	`isZero` is value zero
`parameterCompare`	check largest parameter of self and argument
`this`	pointer to `class` instance called from
`trm`	parameter comparing against
`termEffIndexCK`	minimum of `trm.`κ, `trm.`δ if `trm.`$\delta > 0$ otherwise κ
X	exit code (see Note 36 on page 34)

Table 27: Symbols used in `compare` Table 28

See Table 27 for symbol definitions.		
Comparisons use δ context for both operands.		
Value is `ord.compare(trm)` : -1, 0 or 1 if `ord` $<, =, >$ `trm`		
X	**Condition**	**Value**
	`trm.codeLevel < admisCodeLevel`	
A2	`trm.maxParameter.isZ`	1
A3	`maxParamFirstTerm` $\geq$ `*this` (ignf)	-1
A4		1
A5	`(diff = parameterCompare(trm))`$\neq 0$	diff
A1	`trm.codeLevel > admisCodeLevel`	
	`diff=-trm.compare(*this)`	`diff`
B	$\delta_{ef} \neq 0 \wedge$ `trm.`$\delta_{ef} \neq 0 \wedge$ `((diff=`δ_{ef}`.compare(trm.`δ_{ef}`)` $\neq 0$)	`diff`
B1	`(diff=effIndexCk.compare(termEffIndexCK))` $\neq 0$	`diff`
B1A	`(diff=indexCK.compare(trm.indexCK))` $\neq 0$	`diff`
	`!drillDown.isZ && !trm.drillDown.isZ`	
B2	`(diff=drillDown.compare(trm.drillDown))` $\neq 0$	`diff`
B3	`!drillDown.isZ && trm.drillDown.isZ`	-1
B4	`drillDown.isZ && !trm.drillDown.isZ`	1
A6	`(diff=functionLevel.compare(trm.functionLevel))` $\neq 0$	`diff`
R	`(diff=compareFiniteParams(trm))` $\neq 0$	`diff`
S	`ignoreFactor`	0
S	`(diff=cmp(factor,trm.factor)` $\vee$ `true`	`diff`

Table 28: `AdmisNormalElement::compare` summary

Symbol	**Meaning**
$\delta_{\texttt{ck}}$	if $\kappa = \delta$ use $\delta - 1$
`ddLe`	`drillDownLimitElement` (Table 31)
`dRepl`	copy ordinal called from replacing η parameter
`embLe`	`embedLimitElement` (Table 32)
info	`LimitTypeInfo enum` assigned to this limit
`IFNE::leCom`	`IterFuncNormalElement::limitElementCom` (Section 7.2)
`isLm`	`isLimit`
`isSc`	`isSuccessor`
`indexCKtoLimitType`	computes `limitType` from δ, κ, η
`isZ`	`isZero` is value zero
`le`	`limitElement` (Table 30)
`lEmb(e)`	`limitMaxEmbed(limit)` insure $\delta <$ `limit` by removing δ
α`.leUse(n)`	Compute a safe value for `le(n)`, see Note 47 on page 52
`lme`	`limitMaxEmbed` limit maximum $\delta <$ global δ
`lSg`	value of least significant nonzero β_i
`lsx`	index of least significant nonzero β_i
`lo`	`limitOrd` (Table 39)
α`.loUse(n)`	Computes a safe value for `lo(ord)`, see Note 48 on page 62
`lp1`	`limPlus_1`(Section 6.3)
`nlSg`	value of next to least significant nonzero β_i
`nlsx`	index of next to least significant nonzero β_i
`rep1`	`replace1`, `rep1(i,val)` replaces β_i with `val` in `ord` see Table 9
`rep2`	`replace2`, a two parameter version of `rep1` see Table 9
`Rtn x`	`x` is return value
`sz`	number of β_i in equations 8 and 11
`this`	pointer to `class` instance called from
X	exit code (Note 36 on page 34)

Table 29: Symbols used in `limitElement` Tables 30 to 32 and `limitOrd` Table 39

α is a notation from one of equations 8 to 11.

$\alpha = \omega_{\kappa,\gamma}(\beta_1, \beta_2, ..., \beta_m)$ $\alpha = \omega_\kappa[\eta]$ $\alpha = [[\delta]]\omega_\kappa[\eta]$ $\alpha = [[\delta]]\omega_{\kappa,\gamma}(\beta_1, \beta_2, ..., \beta_m)$

See Table 29 for symbol definitions.

X	**Condition(s)**	LimitTypeInfo	α.le(n)
DDA	!η.isZ		ddLe(n)
LA	lSg.isLm	paramLimit	α.rep1(i,β_i.le(n).lp1())
LB	$\forall_i \beta_i = 0 \wedge \gamma$.isLm	iterLimit	$[[\delta]]\omega_{\kappa,\gamma\texttt{.le(n).lp1}}$
EDDC	κ.isLm $\wedge$ γ.isZ $\wedge$!δ.isZ		embLe(n)
LC	κ.isLm $\wedge$ γ.isZ $\wedge$ δ.isZ	indexCKlimit	$\omega_{\kappa\texttt{.le(ne).lp1()}}$
 CLLM CKLM	sz=1 $\wedge \gamma$.isZ $\wedge$ β_1.isSc κ.isLm $\wedge$ $\kappa = \delta$ κ.isLm $\wedge$ $\kappa \neq \delta$	 indexCKlimitParam indexCKlimitParam	e=κ.le(n) $[[\texttt{e}]]\omega_{\texttt{e},(\omega_\kappa((\beta_1-1)\texttt{.lEmb(e)})).\texttt{lp1}}$ $[[\delta]]\omega_{\texttt{this->leUse(n)},\omega_\kappa(\beta_1-1).\texttt{lp1}}$

CKSC
	sz $= 1 \wedge \gamma$.isZ $\wedge$
	β_1.isSc $\wedge$ κ.isSc
		
		indexCKsuccParam
			b= $[[\delta]]\omega_\kappa(\beta_1 - 1)$.lme
			for(i=1;i<n;i++)
			    b=$[[\delta_{\texttt{ck}}]]\omega_{\kappa-1,\texttt{b.lp1}}(\beta_1 - 1)$
			Rtn b

 KA	γ, isSc $\vee$ sz > 1 $\vee$ $((\gamma > 0) \wedge (\texttt{sz} = 1))$		 IFNE::leCom(n)
 KEA KE	$\forall_i \beta_i = 0 \wedge\ \gamma = 0 \wedge\ \kappa$.isSc κ.isSc $\wedge$ $\delta > 0$ κ.isSc $\wedge$ $\delta = 0$	 indexCKsucc indexCKsucc	 embLe(n) $\omega_\kappa[n]$

Table 30: AdmisNormalElement::limitElement cases

α is a notation from equations 9 or 10.

$\alpha = \omega_\kappa[\eta]$ $\alpha = [[\delta]]\omega_\kappa[\eta]$

The δ parameter is unchanged and not displayed below.

See Table 29 for symbol definitions.

X	**Condition(s)**	LimitTypeInfo	α.le(n)
DDL	η.isLm	drillDownLimit	$\omega_\kappa[\eta\texttt{.limitElement(n)}]$
DDKO	η.isOne $\wedge$ κ.isOne	drillDownSucc	b=ω ;for(i=1;i<n;i++)b=$\varphi_{\texttt{b.lp1}}$;Rtn b
DDO	η.isOne $\wedge$ $\kappa > 1$	drillDownSucc	b= $\omega_{\kappa-1}$;for(i=1;i<n;i++)b=$\omega_{\kappa-1,\texttt{b.lp1}}$

DDGA
	$\eta > 1 \wedge$ κ.isOne	drillDownSucc
		drillDownSucc
			b=dRepl($\eta - 1$);
			for(i=1;i<n;i++)b=$\varphi_{\texttt{b.lp1}}$;Rtn b

DDGB
	$\eta > 1 \wedge$ $\kappa > 1$	drillDownSucc	b=dRepl($\eta - 1$);
			for(i=1;i<n;i++)b=$\omega_{\kappa-1,\texttt{b.lp1}}$;Rtn b

Table 31: AdmisNormalElement::drillDownlimitElement cases

α **is a notation from equation 10 or 11.** $\alpha = [[\delta]]\omega_{\kappa}[\eta]$ $\quad \alpha = [[\delta]]\omega_{\kappa,\gamma}(\beta_1, \beta_2, ..., \beta_m)$.
See Table 29 for symbol definitions.

X	**Condition(s)**	α.`le(n)`
EDEQ	δ`.isLm` $\wedge\ \kappa = \delta$	$\omega_{\kappa.\texttt{le(n)}}[[\delta.\texttt{le(n)}]]$
EDAB	δ`.isLm` $\wedge\ \kappa$`.isLm` $\wedge\ \delta \neq \kappa$	$\omega_{\texttt{leUse}(\kappa.\texttt{le(n)},\delta).\texttt{lp1}}[[\delta]]$
EDAC	κ`.isSc` $\wedge\ \delta$`.isLm`	`b=`$[[\delta]]\omega_{[[\delta]](\omega_{\kappa-1}).\texttt{lp1}}$ `for(i=1;i<n;i++)b=`$[[\delta]]\omega_{\kappa-1,\texttt{b.lp1}}$`;Rtn b`
EDAD	δ`.isSc` $\wedge\ \kappa$`.isLm`	$[[\delta]]\omega_{(\kappa.\texttt{le(n)}+\delta).\texttt{lp1}}$
EDEO	$\delta = 1 \wedge\ \kappa = 1$	`b=`$[[1]]\omega_1$ `for(i=1;i<n;i++)b=`ω_1`[b];Rtn b`
EDEE	$\delta = \kappa\ \wedge\ \delta$`.isSc`	`b=`$\omega_{\kappa}[\omega_{\kappa-1}]$ `for(i=1;i<n;i++)b=`ω_{κ}`[b];Rtn b`
EDEF	$\delta = 1 \wedge\ \kappa > 1 \wedge\ \kappa$`.isSc`	`b=`$[[1]]\omega_{\kappa-1}$ `for(i=1;i<n;i++)b=`$[[1]]\omega_{\kappa-1}$`[b.lp1];Rtn b`
EDEG	$\delta > \kappa \wedge\ \delta$`.isSc` $\wedge\ \kappa$`.isSc`	`b=`$[[\delta-1]]\omega_{\kappa-1,([[delta-1]]\omega_{\kappa}).\texttt{lp1}}$ `for(i=1;i<n;i++)b=`$[[\delta=1]]\omega_{\kappa-1,\texttt{b.lp}}$`;Rtn b`

Table 32: `AdmisNormalElement::embedLimitElement` cases

`AdmisLevOrdinal`	`limitElement`		
ω	1	2	3
ω_ω	ω_1	ω_2	ω_3
ω_1	$\omega_1[1]$	$\omega_1[2]$	$\omega_1[3]$
$\varphi(\omega_1,1,0,0)$	$\varphi(\omega_1,0,1,0)$	$\varphi(\omega_1,0,\varphi(\omega_1,0,1,0)+1,0)$	$\varphi(\omega_1,0,\varphi(\omega_1,0,\varphi(\omega_1,0,1,0)+1,0)+1,0)$
$\varphi(\omega_1+1,1)$	$\varphi(\omega_1+1,0)+1$	$\varphi(\omega_1,\varphi(\omega_1+1,0)+1)$	$\varphi(\omega_1,\varphi(\omega_1,\varphi(\omega_1+1,0)+1)+1)$
$\omega_1[1]$	ω	φ_ω	$\varphi_{\varphi_\omega+1}$
$\omega_1[2]$	$\omega_1[1]$	$\varphi_{\omega_1[1]+1}$	$\varphi_{\varphi_{\omega_1[1]+1}+1}$
$\omega_1[\omega]$	$\omega_1[1]$	$\omega_1[2]$	$\omega_1[3]$
$\omega_2[\omega_1]$	$\omega_2[\omega_1[1]]$	$\omega_2[\omega_1[2]]$	$\omega_2[\omega_1[3]]$
$\omega_2[\omega_1]$	$\omega_2[\omega_1[1]]$	$\omega_2[\omega_1[2]]$	$\omega_2[\omega_1[3]]$
$\omega_2[\omega_2[\omega_1]]$	$\omega_2[\omega_2[\omega_1[1]]]$	$\omega_2[\omega_2[\omega_1[2]]]$	$\omega_2[\omega_2[\omega_1[3]]]$
$(\omega_1 4)$	$(\omega_1 3)+\omega_1[1]$	$(\omega_1 3)+\omega_1[2]$	$(\omega_1 3)+\omega_1[3]$
$\omega^{(\omega_1 2)}$	$\omega^{\omega_1+\omega_1[1]}$	$\omega^{\omega_1+\omega_1[2]}$	$\omega^{\omega_1+\omega_1[3]}$
$\omega_1(1)$	φ_{ω_1+1}	$\varphi_{\varphi_{\omega_1+1}+1}$	$\varphi_{\varphi_{\varphi_{\omega_1+1}+1}+1}$
$\omega_1(1,1)$	$\omega_1(1,0)+1$	$\omega_1(\omega_1(1,0)+1)$	$\omega_1(\omega_1(\omega_1(1,0)+1)+1)$
$\omega_1(1,0,1)$	$\omega_1(1,0,0)+1$	$\omega_1(\omega_1(1,0,0)+1,1)$	$\omega_1(\omega_1(\omega_1(1,0,0)+1,1)+1,1)$
$\omega_1(1,1,0)$	$\omega_1(1,0,1)$	$\omega_1(1,0,\omega_1(1,0,1)+1)$	$\omega_1(1,0,\omega_1(1,0,\omega_1(1,0,1)+1)+1)$
$\omega_1(\varepsilon_0,0)$	$\omega_1(\omega,0)$	$\omega_1(\omega^\omega,0)$	$\omega_1(\omega^{\omega^\omega},0)$
$\omega_1(\varepsilon_0,1,0)$	$\omega_1(\varepsilon_0,0,1)$	$\omega_1(\varepsilon_0,0,\omega_1(\varepsilon_0,0,1)+1)$	$\omega_1(\varepsilon_0,0,\omega_1(\varepsilon_0,0,\omega_1(\varepsilon_0,0,1)+1)+1)$
$\omega_1(\varepsilon_0,1)$	$\omega_1(\omega,\omega_1(\varepsilon_0,0)+1)$	$\omega_1(\omega^\omega,\omega_1(\varepsilon_0,0)+1)$	$\omega_1(\omega^{\omega^\omega},\omega_1(\varepsilon_0,0)+1)$

Table 33: `AdmisLevOrdinal::limitElement` examples part 1.

`AdmisLevOrdinal`	`limitElement`		
ω	1	2	3
$\omega_{1,1}(1)$	$\omega_1(\omega_{1,1}+1)$	$\omega_1(\omega_{1,1}+1,0)$	$\omega_1(\omega_{1,1}+1,0,0)$
$\omega_{2,1}(1)$	$\omega_2(\omega_{2,1}+1)$	$\omega_2(\omega_{2,1}+1,0)$	$\omega_2(\omega_{2,1}+1,0,0)$
$\omega_{1,3}(1)$	$\omega_{1,2}(\omega_{1,3}+1)$	$\omega_{1,2}(\omega_{1,3}+1,0)$	$\omega_{1,2}(\omega_{1,3}+1,0,0)$
$\omega_{1,3}(5)$	$\omega_{1,2}(\omega_{1,3}(4)+1)$	$\omega_{1,2}(\omega_{1,3}(4)+1,0)$	$\omega_{1,2}(\omega_{1,3}(4)+1,0,0)$
$\omega_{1,1}(1,0)$	$\omega_{1,1}(1)$	$\omega_{1,1}(\omega_{1,1}(1)+1)$	$\omega_{1,1}(\omega_{1,1}(\omega_{1,1}(1)+1)+1)$
$\omega_{1,3}(1,0)$	$\omega_{1,3}(1)$	$\omega_{1,3}(\omega_{1,3}(1)+1)$	$\omega_{1,3}(\omega_{1,3}(\omega_{1,3}(1)+1)+1)$
$\omega_{1,3}(5,0)$	$\omega_{1,3}(4,1)$	$\omega_{1,3}(4,\omega_{1,3}(4,1)+1)$	$\omega_{1,3}(4,\omega_{1,3}(4,\omega_{1,3}(4,1)+1)+1)$
$\omega_{1,1}(1,0,0)$	$\omega_{1,1}(1,0)$	$\omega_{1,1}(\omega_{1,1}(1,0)+1,0)$	$\omega_{1,1}(\omega_{1,1}(\omega_{1,1}(1,0)+1,0)+1,0)$
$\omega_{1,3}(1,0,0)$	$\omega_{1,3}(1,0)$	$\omega_{1,3}(\omega_{1,3}(1,0)+1,0)$	$\omega_{1,3}(\omega_{1,3}(\omega_{1,3}(1,0)+1,0)+1,0)$
$\omega_{1,3}(2,0,0)$	$\omega_{1,3}(1,1,0)$	$\omega_{1,3}(1,\omega_{1,3}(1,1,0)+1,0)$	$\omega_{1,3}(1,\omega_{1,3}(1,\omega_{1,3}(1,1,0)+1,0)+1,0)$
$\omega_{1,\varepsilon_0}(1)$	$\omega_{1,\omega}(\omega_{1,\varepsilon_0}+1)$	$\omega_{1,\omega^\omega}(\omega_{1,\varepsilon_0}+1)$	$\omega_{1,\omega^{\omega^\omega}}(\omega_{1,\varepsilon_0}+1)$
$\omega_{1,\varepsilon_0+1}(1)$	$\omega_{1,\varepsilon_0}(\omega_{1,\varepsilon_0+1}+1)$	$\omega_{1,\varepsilon_0}(\omega_{1,\varepsilon_0+1}+1,0)$	$\omega_{1,\varepsilon_0}(\omega_{1,\varepsilon_0+1}+1,0,0)$
$\omega_{1,\varepsilon_0+\omega}(3)$	$\omega_{1,\varepsilon_0+1}(\omega_{1,\varepsilon_0+\omega}(2)+1)$	$\omega_{1,\varepsilon_0+2}(\omega_{1,\varepsilon_0+\omega}(2)+1)$	$\omega_{1,\varepsilon_0+3}(\omega_{1,\varepsilon_0+\omega}(2)+1)$
$\omega_{1,1}(1)$	$\omega_1(\omega_{1,1}+1)$	$\omega_1(\omega_{1,1}+1,0)$	$\omega_1(\omega_{1,1}+1,0,0)$
$\omega_{1,\varepsilon_0}(1)$	$\omega_{1,\omega}(\omega_{1,\varepsilon_0}+1)$	$\omega_{1,\omega^\omega}(\omega_{1,\varepsilon_0}+1)$	$\omega_{1,\omega^{\omega^\omega}}(\omega_{1,\varepsilon_0}+1)$
$\omega_{1,\varepsilon_0+\omega}(1)$	$\omega_{1,\varepsilon_0+1}(\omega_{1,\varepsilon_0+\omega}+1)$	$\omega_{1,\varepsilon_0+2}(\omega_{1,\varepsilon_0+\omega}+1)$	$\omega_{1,\varepsilon_0+3}(\omega_{1,\varepsilon_0+\omega}+1)$
$\omega_{1,\varepsilon_0}(5)$	$\omega_{1,\omega}(\omega_{1,\varepsilon_0}(4)+1)$	$\omega_{1,\omega^\omega}(\omega_{1,\varepsilon_0}(4)+1)$	$\omega_{1,\omega^{\omega^\omega}}(\omega_{1,\varepsilon_0}(4)+1)$
$\omega_{1,\varepsilon_0+1}(5)$	$\omega_{1,\varepsilon_0}(\omega_{1,\varepsilon_0+1}(4)+1)$	$\omega_{1,\varepsilon_0}(\omega_{1,\varepsilon_0+1}(4)+1,0)$	$\omega_{1,\varepsilon_0}(\omega_{1,\varepsilon_0+1}(4)+1,0,0)$
$\omega_{1,\varepsilon_0}(5)$	$\omega_{1,\omega}(\omega_{1,\varepsilon_0}(4)+1)$	$\omega_{1,\omega^\omega}(\omega_{1,\varepsilon_0}(4)+1)$	$\omega_{1,\omega^{\omega^\omega}}(\omega_{1,\varepsilon_0}(4)+1)$
$\omega_1(1,1)$	$\omega_1(1,0)+1$	$\omega_1(\omega_1(1,0)+1)$	$\omega_1(\omega_1(\omega_1(1,0)+1)+1)$
$\omega_1(2)$	$\varphi_{\omega_1(1)+1}(1)$	$\varphi_{\varphi_{\omega_1(1)+1}(1)+1}(1)$	$\varphi_{\varphi_{\varphi_{\omega_1(1)+1}(1)+1}(1)+1}(1)$
$\omega_{1,1}$	$\omega_1(\omega_1+1)$	$\omega_1(\omega_1+1,0)$	$\omega_1(\omega_1+1,0,0)$
$\omega_{1,1}(1)$	$\omega_1(\omega_{1,1}+1)$	$\omega_1(\omega_{1,1}+1,0)$	$\omega_1(\omega_{1,1}+1,0,0)$
$\omega_{1,3}$	$\omega_{1,2}(\omega_{1,2}+1)$	$\omega_{1,2}(\omega_{1,2}+1,0)$	$\omega_{1,2}(\omega_{1,2}+1,0,0)$
$\omega_{1,1}(\omega,1)$	$\omega_{1,1}(1,\omega_{1,1}(\omega,0)+1)$	$\omega_{1,1}(2,\omega_{1,1}(\omega,0)+1)$	$\omega_{1,1}(3,\omega_{1,1}(\omega,0)+1)$
$\omega_{1,\omega}(1)$	$\omega_{1,1}(\omega_{1,\omega}+1)$	$\omega_{1,2}(\omega_{1,\omega}+1)$	$\omega_{1,3}(\omega_{1,\omega}+1)$

Table 34: `AdmisLevOrdinal::limitElement` examples part 2.

`AdmisLevOrdinal`	`limitElement`		
ω	1	2	3
ω^{ω_1+1}	ω_1	$(\omega_1 2)$	$(\omega_1 3)$
$(\omega_1 2)$	$\omega_1+\omega_1[1]$	$\omega_1+\omega_1[2]$	$\omega_1+\omega_1[3]$
$\omega_{\varepsilon_0}(1)$	$\omega_{\omega,\omega_{\varepsilon_0}+1}$	$\omega_{\omega^\omega,\omega_{\varepsilon_0}+1}$	$\omega_{\omega^{\omega^\omega},\omega_{\varepsilon_0}+1}$
$\omega_{\varepsilon_0}(2)$	$\omega_{\omega,\omega_{\varepsilon_0}(1)+1}$	$\omega_{\omega^\omega,\omega_{\varepsilon_0}(1)+1}$	$\omega_{\omega^{\omega^\omega},\omega_{\varepsilon_0}(1)+1}$
$\omega_{\varepsilon_0}(\omega)$	$\omega_{\varepsilon_0}(1)$	$\omega_{\varepsilon_0}(2)$	$\omega_{\varepsilon_0}(3)$
$\omega_{\varepsilon_0+1}(1)$	$\omega_{\varepsilon_0,\omega_{\varepsilon_0+1}+1}$	$\omega_{\varepsilon_0,\omega_{\varepsilon_0,\omega_{\varepsilon_0+1}+1}+1}$	$\omega_{\varepsilon_0,\omega_{\varepsilon_0,\omega_{\varepsilon_0,\omega_{\varepsilon_0+1}+1}+1}+1}$
$\omega_{\varepsilon_0,\omega}+\omega^{\omega_1+1}$	$\omega_{\varepsilon_0,\omega}+\omega_1$	$\omega_{\varepsilon_0,\omega}+(\omega_1 2)$	$\omega_{\varepsilon_0,\omega}+(\omega_1 3)$
$\omega_{1,\omega}(1)$	$\omega_{1,1}(\omega_{1,\omega}+1)$	$\omega_{1,2}(\omega_{1,\omega}+1)$	$\omega_{1,3}(\omega_{1,\omega}+1)$
$\omega_{1,\omega}(\omega)$	$\omega_{1,\omega}(1)$	$\omega_{1,\omega}(2)$	$\omega_{1,\omega}(3)$
$\omega_{1,2}(\varepsilon_0)$	$\omega_{1,2}(\omega)$	$\omega_{1,2}(\omega^\omega)$	$\omega_{1,2}(\omega^{\omega^\omega})$
$\omega_{1,1}(2)$	$\omega_1(\omega_{1,1}(1)+1)$	$\omega_1(\omega_{1,1}(1)+1,0)$	$\omega_1(\omega_{1,1}(1)+1,0,0)$
$\omega_{1,1}(\varepsilon_0)$	$\omega_{1,1}(\omega)$	$\omega_{1,1}(\omega^\omega)$	$\omega_{1,1}(\omega^{\omega^\omega})$
$\omega_{1,1}(1,0,0)$	$\omega_{1,1}(1,0)$	$\omega_{1,1}(\omega_{1,1}(1,0)+1,0)$	$\omega_{1,1}(\omega_{1,1}(\omega_{1,1}(1,0)+1,0)+1,0)$
$\omega_{1,3}(1,0,0)$	$\omega_{1,3}(1,0)$	$\omega_{1,3}(\omega_{1,3}(1,0)+1,0)$	$\omega_{1,3}(\omega_{1,3}(\omega_{1,3}(1,0)+1,0)+1,0)$
$\omega_{1,1}(\omega,1)$	$\omega_{1,1}(1,\omega_{1,1}(\omega,0)+1)$	$\omega_{1,1}(2,\omega_{1,1}(\omega,0)+1)$	$\omega_{1,1}(3,\omega_{1,1}(\omega,0)+1)$
$\omega_{1,1}(\omega,1,0)$	$\omega_{1,1}(\omega,0,1)$	$\omega_{1,1}(\omega,0,\omega_{1,1}(\omega,0,1)+1)$	$\omega_{1,1}(\omega,0,\omega_{1,1}(\omega,0,\omega_{1,1}(\omega,0,1)+1)+1)$
ω_{1,ε_0}	$\omega_{1,\omega}$	ω_{1,ω^ω}	$\omega_{1,\omega^{\omega^\omega}}$
$\omega_{1,\Gamma_0}(1,0)$	$\omega_{1,\Gamma_0}(1)$	$\omega_{1,\Gamma_0}(\omega_{1,\Gamma_0}(1)+1)$	$\omega_{1,\Gamma_0}(\omega_{1,\Gamma_0}(\omega_{1,\Gamma_0}(1)+1)+1)$
$\omega_{1,\Gamma_0}(\varepsilon_0)$	$\omega_{1,\Gamma_0}(\omega)$	$\omega_{1,\Gamma_0}(\omega^\omega)$	$\omega_{1,\Gamma_0}(\omega^{\omega^\omega})$
$\omega_{1,\Gamma_0}(\varepsilon_0,0)$	$\omega_{1,\Gamma_0}(\omega,0)$	$\omega_{1,\Gamma_0}(\omega^\omega,0)$	$\omega_{1,\Gamma_0}(\omega^{\omega^\omega},0)$
$\omega_{1,\Gamma_0}(\varepsilon_0,1)$	$\omega_{1,\Gamma_0}(\omega,\omega_{1,\Gamma_0}(\varepsilon_0,0)+1)$	$\omega_{1,\Gamma_0}(\omega^\omega,\omega_{1,\Gamma_0}(\varepsilon_0,0)+1)$	$\omega_{1,\Gamma_0}(\omega^{\omega^\omega},\omega_{1,\Gamma_0}(\varepsilon_0,0)+1)$
$\omega_{1,\Gamma_0}(\varepsilon_0,1,1)$	$\omega_{1,\Gamma_0}(\varepsilon_0,1,0)+1$	$\omega_{1,\Gamma_0}(\varepsilon_0,0,\omega_{1,\Gamma_0}(\varepsilon_0,1,0)+1)$	$\omega_{1,\Gamma_0}(\varepsilon_0,0,\omega_{1,\Gamma_0}(\varepsilon_0,0,\omega_{1,\Gamma_0}(\varepsilon_0,1,0)+1)+1)$
$\omega_{1,1}$	$\omega_1(\omega_1+1)$	$\omega_1(\omega_1+1,0)$	$\omega_1(\omega_1+1,0,0)$
$\omega_{1,3}$	$\omega_{1,2}(\omega_{1,2}+1)$	$\omega_{1,2}(\omega_{1,2}+1,0)$	$\omega_{1,2}(\omega_{1,2}+1,0,0)$
$\omega_{1,4}$	$\omega_{1,3}(\omega_{1,3}+1)$	$\omega_{1,3}(\omega_{1,3}+1,0)$	$\omega_{1,3}(\omega_{1,3}+1,0,0)$

Table 35: `AdmisLevOrdinal::limitElement` examples part 3.

`AdmisLevOrdinal`	`limitType`	`maxLimitType`
ω_ω	integerLimitType (1)	ω
ω_1	recOrdLimitType (2)	recOrdLimitType (2)
$\varphi(\omega_1, 1, 0, 0)$	recOrdLimitType (2)	recOrdLimitType (2)
$\varphi(\omega_1 + 1, 1)$	integerLimitType (1)	recOrdLimitType (2)
$\omega_1[1]$	integerLimitType (1)	integerLimitType (1)
$\omega_1[2]$	integerLimitType (1)	integerLimitType (1)
$\omega_1[\omega]$	integerLimitType (1)	integerLimitType (1)
$\omega_2[\omega_1]$	recOrdLimitType (2)	recOrdLimitType (2)
$\omega_2[\omega_1]$	recOrdLimitType (2)	recOrdLimitType (2)
$\omega_2[\omega_2[\omega_1]]$	recOrdLimitType (2)	recOrdLimitType (2)
$(\omega_1 4)$	recOrdLimitType (2)	recOrdLimitType (2)
$\omega^{(\omega_1 2)}$	recOrdLimitType (2)	recOrdLimitType (2)
$\omega_1(1)$	integerLimitType (1)	recOrdLimitType (2)
$\omega_1(1, 1)$	integerLimitType (1)	recOrdLimitType (2)
$\omega_1(1, 0, 1)$	integerLimitType (1)	recOrdLimitType (2)
$\omega_1(1, 1, 0)$	integerLimitType (1)	recOrdLimitType (2)
$\omega_1(\varepsilon_0, 0)$	integerLimitType (1)	recOrdLimitType (2)
$\omega_1(\varepsilon_0, 1, 0)$	integerLimitType (1)	recOrdLimitType (2)
$\omega_1(\varepsilon_0, 1)$	integerLimitType (1)	recOrdLimitType (2)

Table 36: `AdmisLevOrdinal limitType` and `maxLimitType` examples part 1.

9.3 `AdmisNormalElement::limitType` member function

For admissible level ordinals, `limitElement` must be supplemented with `limitOrd` that accepts an arbitrary `Ordinal` as input. See Section 8.1 for the reasons behind `limitOrd` and `limitType`. For a particular use of α`.limitOrd(`β`)`, it must be true that β`.maxLimitType` $<$ α`.limitType`. `limitType` and `maxLimitType` both return an `OrdinalImpl`. The routine that computes `limitType` also computes the `enum LimitTypeInfo` that labels limits and is used by `limitElement` and `limitOrd`.

The `limitType` of an ordinal is the `limitType` of the least significant term. Thus `Ordinal::limitType` calls the appropriate `virtual` function such as `AdmisNormalElement::limitType` to do the bulk of the work of `limitType`. Table 38 gives the algorithm used in `AdmisNormalElement::limitType`.

9.4 `AdmisNormalElement::maxLimitType` member function

`maxLimitType` is defined recursively. It is the maximum of of `limitType` and `maxLimitType` for the ordinal and every parameter used in defining the ordinal. This is the complete definition of `maxLimitType` for all base `class`es of `AdmisLevOrdinal`. The δ parameter puts a constraining context on the recursive evaluation and the η parameter can also modify it.

`AdmisNormalElement::maxLimitType`s calls `IterFuncNormalElement::maxLimitType` for the maximum of all parameters except those unique to `class AdmisLevOrdinal`. Next, the type of ω_κ is computed and the maximum of these two values is taken. Finally, the effect

`AdmisLevOrdinal`	`limitType`	`maxLimitType`
ω^{ω_1+1}	integerLimitType (1)	recOrdLimitType (2)
$(\omega_1 2)$	recOrdLimitType (2)	recOrdLimitType (2)
$\omega_{\varepsilon_0}(1)$	integerLimitType (1)	ε_0
$\omega_{\varepsilon_0}(2)$	integerLimitType (1)	ε_0
$\omega_{\varepsilon_0}(\omega)$	integerLimitType (1)	ε_0
$\omega_{\varepsilon_0+1}(1)$	integerLimitType (1)	$\varepsilon_0 + 1$
$\omega_{\varepsilon_0,\omega} + \omega^{\omega_1+1}$	integerLimitType (1)	ε_0
$\omega_{1,\omega}(1)$	integerLimitType (1)	recOrdLimitType (2)
$\omega_{1,\omega}(\omega)$	integerLimitType (1)	recOrdLimitType (2)
$\omega_{1,2}(\varepsilon_0)$	integerLimitType (1)	recOrdLimitType (2)
$\omega_{1,1}(2)$	integerLimitType (1)	recOrdLimitType (2)
$\omega_{1,1}(\varepsilon_0)$	integerLimitType (1)	recOrdLimitType (2)
$\omega_{1,1}(1,0,0)$	integerLimitType (1)	recOrdLimitType (2)
$\omega_{1,3}(1,0,0)$	integerLimitType (1)	recOrdLimitType (2)
$\omega_{1,1}(\omega,1)$	integerLimitType (1)	recOrdLimitType (2)
$\omega_{1,1}(\omega,1,0)$	integerLimitType (1)	recOrdLimitType (2)
ω_{1,ε_0}	integerLimitType (1)	recOrdLimitType (2)
$\omega_{1,\Gamma_0}(1,0)$	integerLimitType (1)	recOrdLimitType (2)
$\omega_{1,\Gamma_0}(\varepsilon_0)$	integerLimitType (1)	recOrdLimitType (2)
$\omega_{1,\Gamma_0}(\varepsilon_0,0)$	integerLimitType (1)	recOrdLimitType (2)
$\omega_{1,\Gamma_0}(\varepsilon_0,1)$	integerLimitType (1)	recOrdLimitType (2)
$\omega_{1,\Gamma_0}(\varepsilon_0,1,1)$	integerLimitType (1)	recOrdLimitType (2)
$\omega_{1,1}$	integerLimitType (1)	recOrdLimitType (2)
$\omega_{1,3}$	integerLimitType (1)	recOrdLimitType (2)
$\omega_{1,4}$	integerLimitType (1)	recOrdLimitType (2)

Table 37: `AdmisLevOrdinal` `limitType` and `maxLimitType` examples part 2.

α is a notation from one of equations 8 to 11.		
$\alpha = \omega_{\kappa,\gamma}(\beta_1, \beta_2, ..., \beta_m)$ $\quad \alpha = \omega_\kappa[\eta]$ $\quad \alpha = [[\delta]]\omega_\kappa[\eta]$ $\quad \alpha = [[\delta]]\omega_{\kappa,\gamma}(\beta_1, \beta_2, ..., \beta_m)$		
See Table 29 for symbol definitions.		
Condition	`LimitTypeInfo`	`limitType`
η`.isLm`	`drillDownLimit`	η`.limitType`
η`.isSc`	`drillDownSuccLimit`	`integerLimitType`
`lSg.isLm`	`paramLimit`	β_i`.limitType`
$\forall_i \beta_i = 0 \wedge \gamma$`.isLm`	`iterLimit`	γ`.limitType`
`lSg.isSc` $\wedge$ `nlSg.isLm`	`paramNxtLimit`	`nlSg.limitType`
`lSg.isSc` $\wedge$ (sz$>$ 1)	`paramSuccLimit`	`integerLimitType`
sz=1 $\wedge \beta_1$.isSc $\wedge\ \gamma$`.isLm`	`iterLimit`	γ`.limitType`
sz=1 $\wedge \beta_1$.isSc $\wedge\ \gamma$`.isSc`	`iterSucc`	`integerLimitType`
sz=1 $\wedge \beta_1$.isSc $\wedge\ \gamma = 0$	`paramSucc`	`integerLimitType`
$\forall_i \beta_i = 0 \wedge \gamma = 0 \wedge \kappa$`.isLm`	`indexCKlimit`	κ`.limitType`
$\forall_i \beta_i = 0 \wedge \gamma = 0 \wedge \kappa$`.isSc`	`indexCKsucc`	`indexCKtoLimitType`

Table 38: `AdmisNormalElement::limitType` description

of the δ and η parameters are taken into account. δ puts an upper bound on `maxLimitType`. A nonzero δ or η reduce the value of `maxLimitType` by 1 if it is a successor. If both are present the value is only reduced by 1.

9.5 `AdmisNormalElement::limitOrd` member function

`AdmisNormalElement::limitOrd` extends the idea of `limitElement` for `limitType`s beyond the integers (`integerLimitType`). An `Ordinal` notation α represents the ordinal that is the union of all ordinals with notations ζ_x such that β_x.`maxLimitType` $< \alpha$.`limitType` $\wedge\ \zeta_x = \alpha$.`limitOrd`(β_x).

`limitOrd` is defined for base classes down to `Ordinal` because instances of those `class`es with appropriate parameters can have `limitType`s greater than `integerLimitType`. For example the expression $\omega^{\omega_1 \times 2}$ defines an `Ordinal` instance with `limitType` $>$ `integerLimitType`.

Table 39 gives the logic of `AdmisNormalElement::limitOrd`. One complication with `limitOrd` occurs when κ is the least significant non zero parameter and δ is nonzero. If $\kappa = \delta$ then one can take `limitOrd` of both values (see the line in Table 39 with exit code LOEG). If not, one must be careful to make sure that the value returned by `limitOrd` does not have $\kappa > \delta$ this is done with routine `loUse`[48].

9.6 `AdmisLevOrdinal::fixedPoint` member function

`AdmisLevOrdinal::fixedPoint` is used by `admisLevelFunctional` to create an instance of

[48]`AdmisNormalElement::loUse` has as its argument the argument to `limitOrd`. It computes `le =` κ`.limitOrd(ord)`. It may then add terms from δ to insure that the value of `le`$\leq \delta$ and insure that `limitOrd` will have a larger output for a larger valid input. All terms in δ that are in `le` or less than terms in `le` *excluding the last term in* `le` are ignored. The remainder are added to `le`.

α is a notation from equations 8 to 11.

$\alpha = \omega_{\kappa,\gamma}(\beta_1, \beta_2, ..., \beta_m)$ $\alpha = \omega_\kappa[\eta]$ $\alpha = [[\delta]]\omega_\kappa[\eta]$ $\alpha = [[\delta]]\omega_{\kappa,\gamma}(\beta_1, \beta_2, ..., \beta_m)$

See Table 29 for symbol definitions.

X	**Condition(s)**	**Info**	α.`limitOrd`(ζ)
LOD	η`.isLm`	`drillDownLimit`	$\omega_{\kappa,\gamma}(\beta_1, \beta_2, .., \beta_n)[\eta.\texttt{lo}(\zeta)]$
LOE	`lSg.isLm`	`paramLimit`	α.`rep1(i,`β_i`.lo(`ζ`))`
LOF	`lSg.isSc nlSg.isLm`	`paramNxtLimit`	`t=rep1(lsx,lSg-1)` `l=nlsig.lo(`ζ`).lp1` `Rtn rep2(lsx,l.lsx+1,t)`
LOED	γ`.isLm` $\wedge$ `sz` $= 1 \wedge$ `lsig.isSc`	`iterNxtLimit`	`t=`$[[\delta]]\omega_\kappa(\texttt{lsig} - 1)$ `l=`γ`.lo(`ζ`).lp1` `Rtn` $[[\delta]]\omega_{\kappa,\texttt{l}}(\texttt{t})$
LOEE	$\forall_i \beta_i = 0 \wedge \gamma$`.isLm`	`iterLimit`	$[[\delta]]\omega_{\kappa,\gamma.\texttt{lo}(\zeta)}$
LOEF	$\forall_i \beta_i = 0 \wedge \gamma, \delta = 0 \wedge \kappa$`.isLm`	`indexCKlimit`	$[[\delta]]\omega_{\kappa.\texttt{lo}(\zeta)}$
LOEG	$\forall_i \beta_i = 0 \wedge \gamma = 0$ $\wedge\ \delta = \kappa \wedge \kappa$`.isLm`	`indexCKlimit`	$[[\delta.\texttt{lo}(\zeta)]]\omega_{\kappa.\texttt{lo}(\zeta)}$
LOEH	$\forall_i \beta_i = 0 \wedge \gamma = 0$ $\wedge\ \delta > 0 \wedge \kappa$`.isLm`	`indexCKlimit`	$[[\delta]]\omega_{\kappa.\texttt{loUse}(\zeta)}$
LOI	$\forall_i \beta_i = 0 \wedge \gamma = 0 \wedge \kappa$`.isLm`	`indexCKSucc`	$[[\delta]]\omega_\kappa[\zeta]$

Table 39: `AdmisNormalElement::limitOrd` cases

`AdmisLevOrdinal`	`limitOrd`		
parameter	ω	ε_0	ω_1
ω_1	$\omega_1[\omega]$	$\omega_1[\varepsilon_0]$	`limType` too large
ω_2	$\omega_2[\omega]$	$\omega_2[\varepsilon_0]$	$\omega_2[\omega_1]$
$(\omega_1 4)$	$(\omega_1 3) + \omega_1[\omega]$	$(\omega_1 3) + \omega_1[\varepsilon_0]$	`limType` too large
$\omega^{(\omega_1 2)}$	$\omega^{\omega_1 + \omega_1[\omega]}$	$\omega^{\omega_1 + \omega_1[\varepsilon_0]}$	`limType` too large
$\omega_{\omega_{\omega_1+1}}(\omega_{\omega_1+1})$	$\omega_{\omega_{\omega_1+1}}(\omega_{\omega_1+1}[\omega])$	$\omega_{\omega_{\omega_1+1}}(\omega_{\omega_1+1}[\varepsilon_0])$	$\omega_{\omega_{\omega_1+1}}(\omega_{\omega_1+1}[\omega_1])$
ω_{ω_1+1}	$\omega_{\omega_1+1}[\omega]$	$\omega_{\omega_1+1}[\varepsilon_0]$	$\omega_{\omega_1+1}[\omega_1]$
ω_{1,ω_1}	$\omega_{1,\omega_1[\omega]}$	$\omega_{1,\omega_1[\varepsilon_0]}$	`limType` too large
ω_{100}	$\omega_{100}[\omega]$	$\omega_{100}[\varepsilon_0]$	$\omega_{100}[\omega_1]$
$\omega_{1,1}(\omega_1)$	$\omega_{1,1}(\omega_1[\omega])$	$\omega_{1,1}(\omega_1[\varepsilon_0])$	`limType` too large
$\omega_{1,3}(\omega_1)$	$\omega_{1,3}(\omega_1[\omega])$	$\omega_{1,3}(\omega_1[\varepsilon_0])$	`limType` too large
$\omega_{\varepsilon_0}(\omega_1)$	$\omega_{\varepsilon_0}(\omega_1[\omega])$	$\omega_{\varepsilon_0}(\omega_1[\varepsilon_0])$	`limType` too large
$\omega_{\varepsilon_0+1}(1)$	`limType` too large	`limType` too large	`limType` too large
$\omega_{\varepsilon_0,\omega} + \omega^{\omega_1+1}$	`limType` too large	`limType` too large	`limType` too large

Table 40: `AdmisLevOrdinal::limitOrd` examples.

an `AdmisLevOrdinal` normal form (Equations 8 to 11) that is the simplest expression for the ordinal represented. The routine has the following parameters.

- The admissible ordinal index or κ from Equation 8.
- The function level or γ from the equation.
- An index specifying the largest parameter of the ordinal notation being constructed. If the largest parameter is the admissible ordinal index this index has the value `indexCKmaxParam` defined as -2 in an `enum`. If the largest parameter is the function level the index has the value `iterMaxParam` defined as -1.
- The function parameters (a `NULL` terminated array of pointers to `Ordinals`) or just `NULL` of there are none. These are the β_j from a term in Equation 7.

This function determines if the parameter, at the specified index, is a fixed point for an `AdmisLevOrdinal` created with the specified parameters. If so, `true` is returned and otherwise `false`. The routine that calls this routine selects the largest parameter from the admissible ordinal index (κ), the function level (γ) and the array of `Ordinal` pointers (β_j) and indicates this in the index parameter, The calling routine checks to see if all less significant parameters are 0. If not, this cannot be a fixed point. Thus `fixedPoint` is called only if this condition is met.

Section 6.3 describes `psuedoCodeLevel`. If the `psuedoCodeLevel` of the selected parameter is less than or equal `cantorCodeLevel`, `false` is returned. If that level is greater than `AdmisCodeLevel`, `true` is returned. The most significant parameter, the admissible level index, cannot be a fixed point unless it has a `psuedoCodeLevel` $>$ `AdmisCodeLevel`. Thus, if the index selects the the admissible level index, and the previous test was not passed `false` is returned. Finally an `AdmisLevOrdinal` is constructed from all the parameters except that selected by the index. If this value is less than the selected parameter, `true` is returned and otherwise `false`.

9.7 `AdmisLevOrdinal` operators

The multiplication and exponentiation. routines for `FiniteFuncOrdinal`, `Ordinal` and the associated `class`es for normal form terms do not need to be overridden except for some utilities such as that used to create a copy of an `AdmisNormalElement` normal form term with a new value for `factor`.

Some multiplication examples are shown in Table 41 Some exponentiation examples are shown in Table 42

10 Philosophical Issues

This approach to the ordinals has its roots in a philosophy of mathematical truth that rejects the Platonic ideal of completed infinite totalities[3, 2]. It replaces the impredictivity inherent in that philosophy with explicit incompleteness. It is a philosophy that interprets Cantor's proof that the reals are not countable as the first major incompleteness theorem. Cantor

α	β	$\alpha \times \beta$
ω_1	ω_1	$\omega^{(\omega_1 2)}$
ω_1	φ_3	$\omega^{\omega_1+\varphi_3}$
ω_1	$\omega_1+[[1]]\omega_{3,1}(1,0,1)$	$\omega^{(\omega_1 2)}+\omega^{\omega_1+[[1]]\omega_{3,1}(1,0,1)}$
ω_1	$\omega_{1,\omega}(\omega,1)$	$\omega_{1,\omega}(\omega,1)$
ω_1	$\omega_{1,\varepsilon_0}(\omega,1,0)$	$\omega_{1,\varepsilon_0}(\omega,1,0)$
ω_1	ω_{ε_0}	ω_{ε_0}
φ_3	ω_1	ω_1
φ_3	φ_3	$\omega^{(\varphi_3 2)}$
φ_3	$\omega_1+[[1]]\omega_{3,1}(1,0,1)$	$\omega_1+[[1]]\omega_{3,1}(1,0,1)$
φ_3	$\omega_{1,\omega}(\omega,1)$	$\omega_{1,\omega}(\omega,1)$
φ_3	$\omega_{1,\varepsilon_0}(\omega,1,0)$	$\omega_{1,\varepsilon_0}(\omega,1,0)$
φ_3	ω_{ε_0}	ω_{ε_0}
$\omega_1+[[1]]\omega_{3,1}(1,0,1)$	ω_1	$\omega^{(\omega_1 2)}$
$\omega_1+[[1]]\omega_{3,1}(1,0,1)$	φ_3	$\omega^{\omega_1+\varphi_3}$
$\omega_1+[[1]]\omega_{3,1}(1,0,1)$	$\omega_1+[[1]]\omega_{3,1}(1,0,1)$	$\omega^{(\omega_1 2)}+\omega^{\omega_1+[[1]]\omega_{3,1}(1,0,1)}$
$\omega_1+[[1]]\omega_{3,1}(1,0,1)$	$\omega_{1,\omega}(\omega,1)$	$\omega_{1,\omega}(\omega,1)$
$\omega_1+[[1]]\omega_{3,1}(1,0,1)$	$\omega_{1,\varepsilon_0}(\omega,1,0)$	$\omega_{1,\varepsilon_0}(\omega,1,0)$
$\omega_1+[[1]]\omega_{3,1}(1,0,1)$	ω_{ε_0}	ω_{ε_0}
$\omega_{1,\omega}(\omega,1)$	ω_1	$\omega^{\omega_{1,\omega}(\omega,1)+\omega_1}$
$\omega_{1,\omega}(\omega,1)$	φ_3	$\omega^{\omega_{1,\omega}(\omega,1)+\varphi_3}$
$\omega_{1,\omega}(\omega,1)$	$\omega_1+[[1]]\omega_{3,1}(1,0,1)$	$\omega^{\omega_{1,\omega}(\omega,1)+\omega_1}+\omega^{\omega_{1,\omega}(\omega,1)+[[1]]\omega_{3,1}(1,0,1)}$
$\omega_{1,\omega}(\omega,1)$	$\omega_{1,\omega}(\omega,1)$	$\omega^{(\omega_{1,\omega}(\omega,1)2)}$
$\omega_{1,\omega}(\omega,1)$	$\omega_{1,\varepsilon_0}(\omega,1,0)$	$\omega_{1,\varepsilon_0}(\omega,1,0)$
$\omega_{1,\omega}(\omega,1)$	ω_{ε_0}	ω_{ε_0}
$\omega_{1,\varepsilon_0}(\omega,1,0)$	ω_1	$\omega^{\omega_{1,\varepsilon_0}(\omega,1,0)+\omega_1}$
$\omega_{1,\varepsilon_0}(\omega,1,0)$	φ_3	$\omega^{\omega_{1,\varepsilon_0}(\omega,1,0)+\varphi_3}$
$\omega_{1,\varepsilon_0}(\omega,1,0)$	$\omega_1+[[1]]\omega_{3,1}(1,0,1)$	$\omega^{\omega_{1,\varepsilon_0}(\omega,1,0)+\omega_1}+\omega^{\omega_{1,\varepsilon_0}(\omega,1,0)+[[1]]\omega_{3,1}(1,0,1)}$
$\omega_{1,\varepsilon_0}(\omega,1,0)$	$\omega_{1,\omega}(\omega,1)$	$\omega^{\omega_{1,\varepsilon_0}(\omega,1,0)+\omega_{1,\omega}(\omega,1)}$
$\omega_{1,\varepsilon_0}(\omega,1,0)$	$\omega_{1,\varepsilon_0}(\omega,1,0)$	$\omega^{(\omega_{1,\varepsilon_0}(\omega,1,0)2)}$
$\omega_{1,\varepsilon_0}(\omega,1,0)$	ω_{ε_0}	ω_{ε_0}
ω_{ε_0}	ω_1	$\omega^{\omega_{\varepsilon_0}+\omega_1}$
ω_{ε_0}	φ_3	$\omega^{\omega_{\varepsilon_0}+\varphi_3}$
ω_{ε_0}	$\omega_1+[[1]]\omega_{3,1}(1,0,1)$	$\omega^{\omega_{\varepsilon_0}+\omega_1}+\omega^{\omega_{\varepsilon_0}+[[1]]\omega_{3,1}(1,0,1)}$
ω_{ε_0}	$\omega_{1,\omega}(\omega,1)$	$\omega^{\omega_{\varepsilon_0}+\omega_{1,\omega}(\omega,1)}$
ω_{ε_0}	$\omega_{1,\varepsilon_0}(\omega,1,0)$	$\omega^{\omega_{\varepsilon_0}+\omega_{1,\varepsilon_0}(\omega,1,0)}$
ω_{ε_0}	ω_{ε_0}	$\omega^{(\omega_{\varepsilon_0}2)}$

Table 41: `AdmisLevOrdinal` multiply examples

α	β	α^{β}
ω_1	ω_1	$\omega^{\omega^{(\omega_1 2)}}$
ω_1	φ_3	$\omega^{\omega^{\omega_1+\varphi_3}}$
ω_1	$\omega_1 + [[1]]\omega_{3,1}(1,0,1)$	$\omega^{\omega^{(\omega_1 2)}+\omega^{\omega_1+[[1]]\omega_{3,1}(1,0,1)}}$
ω_1	$\omega_{1,\omega}(\omega,1)$	$\omega_{1,\omega}(\omega,1)$
ω_1	$\omega_{1,\varepsilon_0}(\omega,1,0)$	$\omega_{1,\varepsilon_0}(\omega,1,0)$
ω_1	ω_{ε_0}	ω_{ε_0}
φ_3	ω_1	ω_1
φ_3	φ_3	$\omega^{\omega^{(\varphi_3 2)}}$
φ_3	$\omega_1 + [[1]]\omega_{3,1}(1,0,1)$	$\omega^{\omega_1+[[1]]\omega_{3,1}(1,0,1)}$
φ_3	$\omega_{1,\omega}(\omega,1)$	$\omega_{1,\omega}(\omega,1)$
φ_3	$\omega_{1,\varepsilon_0}(\omega,1,0)$	$\omega_{1,\varepsilon_0}(\omega,1,0)$
φ_3	ω_{ε_0}	ω_{ε_0}
$\omega_1 + [[1]]\omega_{3,1}(1,0,1)$	ω_1	$\omega^{\omega^{(\omega_1 2)}}$
$\omega_1 + [[1]]\omega_{3,1}(1,0,1)$	φ_3	$\omega^{\omega^{\omega_1+\varphi_3}}$
$\omega_1 + [[1]]\omega_{3,1}(1,0,1)$	$\omega_1 + [[1]]\omega_{3,1}(1,0,1)$	$\omega^{\omega^{(\omega_1 2)}+\omega^{\omega_1+[[1]]\omega_{3,1}(1,0,1)}}$
$\omega_1 + [[1]]\omega_{3,1}(1,0,1)$	$\omega_{1,\omega}(\omega,1)$	$\omega_{1,\omega}(\omega,1)$
$\omega_1 + [[1]]\omega_{3,1}(1,0,1)$	$\omega_{1,\varepsilon_0}(\omega,1,0)$	$\omega_{1,\varepsilon_0}(\omega,1,0)$
$\omega_1 + [[1]]\omega_{3,1}(1,0,1)$	ω_{ε_0}	ω_{ε_0}
$\omega_{1,\omega}(\omega,1)$	ω_1	$\omega^{\omega^{\omega_{1,\omega}(\omega,1)+\omega_1}}$
$\omega_{1,\omega}(\omega,1)$	φ_3	$\omega^{\omega^{\omega_{1,\omega}(\omega,1)+\varphi_3}}$
$\omega_{1,\omega}(\omega,1)$	$\omega_1 + [[1]]\omega_{3,1}(1,0,1)$	$\omega^{\omega^{\omega_{1,\omega}(\omega,1)+\omega_1}+\omega^{\omega_{1,\omega}(\omega,1)+[[1]]\omega_{3,1}(1,0,1)}}$
$\omega_{1,\omega}(\omega,1)$	$\omega_{1,\omega}(\omega,1)$	$\omega^{\omega^{(\omega_{1,\omega}(\omega,1)2)}}$
$\omega_{1,\omega}(\omega,1)$	$\omega_{1,\varepsilon_0}(\omega,1,0)$	$\omega_{1,\varepsilon_0}(\omega,1,0)$
$\omega_{1,\omega}(\omega,1)$	ω_{ε_0}	ω_{ε_0}
$\omega_{1,\varepsilon_0}(\omega,1,0)$	ω_1	$\omega^{\omega^{\omega_{1,\varepsilon_0}(\omega,1,0)+\omega_1}}$
$\omega_{1,\varepsilon_0}(\omega,1,0)$	φ_3	$\omega^{\omega^{\omega_{1,\varepsilon_0}(\omega,1,0)+\varphi_3}}$
$\omega_{1,\varepsilon_0}(\omega,1,0)$	$\omega_1 + [[1]]\omega_{3,1}(1,0,1)$	$\omega^{\omega^{\omega_{1,\varepsilon_0}(\omega,1,0)+\omega_1}+\omega^{\omega_{1,\varepsilon_0}(\omega,1,0)+[[1]]\omega_{3,1}(1,0,1)}}$
$\omega_{1,\varepsilon_0}(\omega,1,0)$	$\omega_{1,\omega}(\omega,1)$	$\omega^{\omega^{\omega_{1,\varepsilon_0}(\omega,1,0)+\omega_{1,\omega}(\omega,1)}}$
$\omega_{1,\varepsilon_0}(\omega,1,0)$	$\omega_{1,\varepsilon_0}(\omega,1,0)$	$\omega^{\omega^{(\omega_{1,\varepsilon_0}(\omega,1,0)2)}}$
$\omega_{1,\varepsilon_0}(\omega,1,0)$	ω_{ε_0}	ω_{ε_0}
ω_{ε_0}	ω_1	$\omega^{\omega^{\omega_{\varepsilon_0}+\omega_1}}$
ω_{ε_0}	φ_3	$\omega^{\omega^{\omega_{\varepsilon_0}+\varphi_3}}$
ω_{ε_0}	$\omega_1 + [[1]]\omega_{3,1}(1,0,1)$	$\omega^{\omega^{\omega_{\varepsilon_0}+\omega_1}+\omega^{\omega_{\varepsilon_0}+[[1]]\omega_{3,1}(1,0,1)}}$
ω_{ε_0}	$\omega_{1,\omega}(\omega,1)$	$\omega^{\omega^{\omega_{\varepsilon_0}+\omega_{1,\omega}(\omega,1)}}$
ω_{ε_0}	$\omega_{1,\varepsilon_0}(\omega,1,0)$	$\omega^{\omega^{\omega_{\varepsilon_0}+\omega_{1,\varepsilon_0}(\omega,1,0)}}$
ω_{ε_0}	ω_{ε_0}	$\omega^{\omega^{(\omega_{\varepsilon_0}2)}}$

Table 42: `AdmisLevOrdinal` exponential examples

proved that any formal system that meets certain minimal requirements must be incomplete, because it can always be expanded by consistently adding more real numbers to it. This can be done, from outside the system, by a Cantor diagonalization of the reals definable within the system.

Because of mathematics' inherent incompleteness, it can always be expanded. Thus it is consistent but, I suspect, incorrect to reason as if completed infinite totalities exist. This does not mean that an algebra of infinities or infinitesimals is not useful. As long as they get the same results is reasoning about the potentially infinite they may be of significant practical value.

The names of all the reals provably definable in any finite (or recursively enumerable) formal system must be recursively enumerable, as Löwenheim and Skolem observed in the theorem that bears their names. Thus one can consistently assume the reals *in a consistent formal system that meets other requirements* form a completed totality, albeit one that is not recursively enumerable *within* the system.

The current philosophical approach to mathematical truth has been enormously successful. This is the most powerful argument in support of it. However, I believe the approach, that was so successful in the past, is increasingly becoming a major obstacle to mathematical progress. If mathematics is about completed infinite totalities, then computer technology is of limited value in expanding the foundations. For computers are restricted to finite operations in contrast to the supposed human ability to transcend the finite through pure thought and mathematical intuition. Thus the foundations of mathematics is perhaps the only major scientific field where computers are not an essential tool for research. An ultimate goal of this research is to help to change that perspective and the practical reality of foundations research in mathematics.

Since all ordinals beyond the integers are infinite they do not correspond to anything in the physical world. Our idea of *all* integers comes from the idea that we can define what an integer is. The property of being an integer leads to the idea that there is a set of *all* objects satisfying the property. An alternative way to think of the integers is computationally. We can write a computer program that can *in theory* output every integer. Of course real programs do not run forever, error free, but that does not mean that such potentially infinite operations as a computer running forever lack physical significance. Our universe appears to be extraordinarily large, but finite. However, it might be potentially infinite. Cosmology is of necessity a speculative science. Thus the idea of a *potentially* infinite set of all integers, in contrast to that of completed infinite totalities, might have objective meaning in physical reality. Most of standard mathematics has an interpretation in an always finite but potentially infinite universe, but some questions, such as the continuum hypothesis do not. This meshes with increasing skepticism about whether the continuum hypothesis and other similar foundations questions are objectively true or false as Solomon Feferman and others have suggested[5].

A Command line interface

This appendix is a stand alone manual for a command line interface to most of the capabilities described in this document.

A.1 Introduction

The Ordinal Calculator is an interactive tool for understanding the hierarchies of recursive and countable ordinals[15, 11, 6]. It is also a research tool to help to expand these hierarchies. Its motivating goal is ultimately to expand the foundations of mathematics by using computer technology to manage the combinatorial explosion in complexity that comes with explicitly defining the recursive ordinals implicitly defined by the axioms of Zermelo-Frankel set theory[4, 2]. The underlying philosophy focuses on what formal systems tell us about physically realizable combinatorial processes[3].

The source code and documentation is licensed for use and and distribution under the Gnu General Public License, Version 2, June 1991. A copy of this license must be distributed with the program. It is also at: `http://www.gnu.org/licenses/gpl-2.0.html`. The ordinal calculator source code, documentation and some executables can be downloaded from: `http://www.mtnmath.com/ord` or `https://sourceforge.net/projects/ord`.

Most of this manual is automatically extracted from the online documentation.

This is a command line interactive interface to a program for exploring the ordinals. It supports recursive ordinals up to and beyond the the Bachmann-Howard ordinal[8]. It defines notations for the Church-Kleene ordinal and some larger countable ordinals. We refer to these as admissible level ordinals. They are used in a form of ordinal collapsing to define large recursive ordinals.

A.2 Command line options

The standard name for the ordinal calculator is `ord`. Typing `ord` (or `./ord`) ENTER will start `ord` in command line mode on most Unix or Linux based systems. The other command line options are mostly for validating or documenting `ord`. They are:

‘`cmd`’ — Read specified command file and enter command line mode.
‘`version`’ — Print program version.
‘`help`’ — Describe command line options.
‘`cmdDoc`’ — Write manual for command line mode in TeX format.
‘`tex`’ — Output TeX documentation files.
‘`psi`’ — Do tests of Veblen hierarchy.
‘`base`’ — Do tests of base class Ordinal.
‘`try`’ — Do tests of class FiniteFuncOrdinal.
‘`iter`’ — Do tests of class IterFuncOrdinal.
‘`admis`’ — Do tests of class AdmisLevOrdinal.

'`admis2`' — Do additional tests of class AdmisLevOrdinal.
'`play`' — Do integrating tests.
'`descend`' — Test descending trees.
'`collapse`' — Ordinal collapsing tests.
'`nested`' — Ordinal nested collapsing tests.
'`nested2`' — Ordinal nested collapsing tests 2.
'`nested3`' — Ordinal nested collapsing tests 3.
'`exitCode`' — LimitElement exit code base test.
'`exitCode2`' — LimitElement exit code base test 2.
'`limitEltExitCode`' — Admissible level LimitElement exit code test 0.
'`limitEltExitCode1`' — Admissible level limitElement exit code test 1.
'`limitEltExitCode2`' — Admissible level limitElement exit code test 2.
'`limitEltExitCode3`' — Admissible level limitElement exit code test 3.
'`limitOrdExitCode`' — Admissible level limitOrd exit code test.
'`limitOrdExitCode1`' — Admissible level limitOrd exit code test.
'`limitOrdExitCode2`' — Admissible level limitOrd exit code test.
'`limitOrdExitCode3`' — Admissible level limitOrd exit code test.
'`transition`' — Admissible level transition test.
'`cmpExitCode`' — Admissible level compare exit code test.
'`drillDownExitCode`' — Admissible level compare exit code test.
'`embedExitCode`' — Admissible level compare exit code test.
'`fixedPoint`' — test fixed point detection.
'`helpTex`' — TeX document command line options.

A.3 Help topics

Following are topics you can get more information about by entering '`help` topic'.
'`cmds`' – lists commands.
'`defined`' – list predefined ordinal variables.
'`compare`' – describes comparison operators.
'`members`' – describes member functions.
'`ordinal`' – describes available ordinal notations.
'`ordlist`' – describes ordinal lists and their use.
'`purpose`' – describes the purpose and philosophy of this project.
'`syntax`' – describes syntax.
'`version`' – displays program version.

This program supports GNU 'readline' input line editing.
You can download the program and documentation at: Mountain Math Software or at SourceForge.net (`http://www.mtnmath.com/ord` or `https://sourceforge.net/projects/ord`).

A.4 Ordinals

Ordinals are displayed in TeX and plain text format. (Enter 'help opts' to control this.) The finite ordinals are the nonnegative integers. The ordinal operators are +, * and ^ for addition, multiplication and exponentiation. Exponentiation has the highest precedence. Parenthesis can be used to group subexpressions.

The ordinal of the integers, `omega`, is represented by the single lowercase letter: '`w`'. The Veblen function is specified as '`psi(p1,p2,...,pn)`' where n is any integer > 0. Special notations are displayed in some cases. Specifically `psi(x)` is displayed as `w^x`. `psi(1,x)` is displayed as `epsilon(x)`. `psi(1,x,0)` is displayed as `gamma(x)`. In all cases the displayed version can be used as input.

Larger ordinals are specified as `psi_{px}(p1,p2,...,pn)`. The first parameter is enclosed in brackets not parenthesis. `psi_{1}` is defined as the union of `w`, `epsilon(0)`, `gamma(0)`, `psi(1, 0, 0, 0)`, `psi(1, 0, 0, 0, 0)`, `psi(1, 0, 0, 0, 0, 0)`, `psi(1, 0, 0, 0, 0, 0, 0)`, ... You can access the sequence whose union is a specific ordinal using member functions. Type `help members` to learn more about this. Larger notations beyond the recursive ordinals are also available in this implementation. See the documentation 'A Computational Approach to the Ordinal Numbers' to learn about 'Countable admissible ordinals'.

There are several predefined ordinals. '`w`' and '`omega`' can be be used interchangeably for the ordinal of the integers and in other contexts. '`eps0`' and '`omega1CK`' are also predefined. Type 'help defined' to learn more.

A.5 Predefined ordinals

The predefined ordinal variables are:
`omega` $= \omega$
`w` $= \omega$
`omega1CK` $= \omega_1$
`w1` $= \omega_1$
`w1CK` $= \omega_1$
`eps0` $= \varepsilon_0$

A.6 Syntax

The syntax is that of restricted arithmetic expressions and assignment statements. The tokens are variable names, nonnegative integers and the operators: +, * and ^ (addition, multiplication and exponentiation). Comparison operators are also supported. Type '`help comparison`' to learn about them. The letter 'w' is predefined as omega, the ordinal of the integers. Type '`help defined`' for a list of all predefined variables. To learn more about ordinals type '`help ordinal`'. C++ style member functions are supported with a '.' separating the variable name (or expression enclosed in parenthesis) from the member function name. Enter '`help members`' for the list of member functions.

An assignment statement or ordinal expression can be entered and it will be evaluated and displayed in normal form. Typing '`help opts`' lists the display options. Assignment statements are stored. They can be listed (command 'list') and their value can be used in subsequent expressions. All statements end at the end of a line unless the last character is '\'. Lines can be continued indefinitely. Comments must be preceded by either '`%`' or '`//`'.

Commands can be entered as one or more names separated by white space. File names should be enclosed in double quotes (`"`) if they contain any non alphanumeric characters such as dot, '.'. Command names can be used as variables. Enter '`help cmds`' to get a list of commands and their functions.

A.7 Ordinal lists

Lists are a sequence of ordinals. An assignment statement can name a single ordinal or a list of them separated by commas. In most circumstances only the first element in the list is used, but some functions (such as member function 'limitOrdLst') use the full list. Type 'help members' to learn more about 'limitOrdLst'.

A.8 Commands

A.8.1 All commands

The following commands are available:
'`cmpCheck`' – toggle comparison checking for debugging.
'`examples`' – shows examples.
'`exit`' – exits the program.
'`exportTeX`' – exports assignments statements in TeX format.
'`help`' – displays information on various topics.
'`list`' – lists assignment statements.
'`log`' – writes a log file (ord.log default).
'`listTeX`' – lists assignment statements in TeX format.
'`logopt`' – controls the log file.
'`opts`' – controls display format and other options.
'`prompt`' – prompts for ENTER with optional string argument.
'`quit`' – exits the program.
'`read`' – read "input file" (ord_calc.ord default).
'`readall`' – same as read but no 'wait for ENTER' prompt.
'`save`' – saves assignment statements to a file (ord_calc.ord default).
'`setDbg`' – set debugging options.
'`yydebug`' – enables parser debugging (off option).

A.8.2 Commands with options

Following are the commands with options.

Command '`examples`' – shows examples.
It has one parameter with the following options.
'`arith`' – demonstrates ordinal arithmetic.
'`compare`' – shows compare examples.
'`display`' – shows how display options work.
'`member`' – demonstrates member functions.
'`VeblenFinite`' – demonstrates Veblen functions of a finite number of ordinals.
'`VeblenExtend`' – demonstrates Veblen functions iterated up to a recursive ordinal.
'`admissible`' – demonstrates countable admissible level ordinal notations.
'`admissibleDrillDown`' – demonstrates admissible notations dropping down one level.
'`admissibleContext`' – demonstrates admissible ordinal context parameters.
'`list`' – shows how lists work.
'`desLimitOrdLst`' – shows how to construct a list of descending trees.

Command '`logopt`' – controls the log file.
It has one parameter with the following options.
'`flush`' – flush log file.
'`stop`' – stop logging.

Command '`opts`' – controls display format and other options.
It has one parameter with the following options.
'`both`' – display ordinals in both plain text and TeX formats.
'`tex`' – display ordinals in TeX format only.
'`text`' – display ordinals in plain text format only (default).
'`psi`' – additionally display ordinals in Psi format (turned off by the above options).
'`promptLimit`' – lines to display before pausing, < 4 disables pause.

Command '`setDbg`' – set debugging options.
It has one parameter with the following options.
'`all`' – turn on all debugging.
'`arith`' – debug ordinal arithmetic.
'`clear`' – turn off all debugging.
'`compare`' – debug compare.
'`exp`' – debug exponential.
'`limArith`' – limited debugging of arithmetic.
'`limit`' – debug limit element functions.
'`construct`' – debug constructors.

A.9 Member functions

Every ordinal (except 0) is the union of smaller ordinals. Every limit ordinal is the union of an infinite sequence of smaller ordinals. Member functions allow access to to these smaller ordinals. One can specify how many elements of this sequence to display or get the value of a specific instance of the sequence. For a limit ordinal, the sequence displayed, were it extended to infinity and its union taken, that union would equal the original ordinal.

The syntax for a member function begins with either an ordinal name (from an assignment statement) or an ordinal expression enclosed in parenthesis. This is followed by a dot (.) and then the member function name and its parameters enclosed in parenthesis. The format is 'ordinal_name.memberFunction(p)' where p may be optional. Functions 'limitOrdLst' and 'desLimitOrdLst' return a list all other member functions return a scalar value. Unless specified otherwise, the returned value is that of the ordinal the function was called from.

The member functions are:

'`descend`' – (n,m) iteratively (up to m) take nth limit element.
'`descendFull`' – (n,m,k) iteratively (up to m) take n limit elements with root k.
'`getCompareIx`' – display admissible compare index.
'`limitElt`' – evaluates to specified finite limit element.
'`limitElement`' – an alias for 'limitElt'.
'`listLimitElts`' – lists specified (default 10) limit elements.
'`listElts`' – alias for listLimitElts.
'`limitOrd`' – evaluates to specified (may be infinite) limit element.
'`limitType`' – return limitType.
'`limitOrdLst`' – apply each input from list to limitOrd and return that list.
'`desLimitOrdLst`' – (depth, list) does limitOrdLst iteratively on all outputs depth times.
'`maxLimitType`' – return maxLimitType.
'`maxParameter`' – return maxParameter (for debugging).

A.10 Comparison operators

Any two ordinals or ordinal expressions can be compared using the operators: <, <=, >, >= and ==. The result of the comparison is the text either TRUE or FALSE. Comparison operators have lower precedence than ordinal operators.

A.11 Examples

In the examples a line that begins with the standard prompt 'ordCalc> ' contains user input. All other lines contain program output

To select an examples type '`examples`' followed by one of the following options.
'`arith`' – demonstrates ordinal arithmetic.
'`compare`' – shows compare examples.

'`display`' – shows how display options work.
'`member`' – demonstrates member functions.
'`VeblenFinite`' – demonstrates Veblen functions of a finite number of ordinals.
'`VeblenExtend`' – demonstrates Veblen functions iterated up to a recursive ordinal.
'`admissible`' – demonstrates countable admissible level ordinal notations.
'`admissibleDrillDown`' – demonstrates admissible notations dropping down one level.
'`admissibleContext`' – demonstrates admissible ordinal context parameters.
'`list`' – shows how lists work.
'`desLimitOrdLst`' – shows how to construct a list of descending trees.

A.11.1 Simple ordinal arithmetic

The following demonstrates ordinal arithmetic.

```
ordCalc> a=w^w
Assigning ( w^w ) to 'a'.
ordCalc> b=w*w
Assigning ( w^2 ) to 'b'.
ordCalc> c=a+b
Assigning ( w^w ) + ( w^2 ) to 'c'.
ordCalc> d=b+a
Assigning ( w^w ) to 'd'.
```

A.11.2 Comparison operators

The following shows compare examples.

```
ordCalc> psi(1,0,0) == gamma(0)
TRUE
ordCalc> psi(1,w) == epsilon(w)
TRUE
ordCalc> w^w < psi(1)
FALSE
ordCalc> psi(1)
Normal form:  w
```

A.11.3 Display options

The following shows how display options work.

```
ordCalc> a=w^(w^w)
Assigning ( w^( w^w ) ) to 'a'.
ordCalc> b=epsilon(a)
```

```
Assigning epsilon( ( w^( w^w ) )) to ‘b’.
ordCalc> c=gamma(b)
Assigning gamma( epsilon( ( w^( w^w ) )) ) to ‘c’.
ordCalc> list
a = ( w^( w^w ) )
b = epsilon( ( w^( w^w ) ))
c = gamma( epsilon( ( w^( w^w ) )) )
ordCalc> opts tex
ordCalc> list
a = \omega{}^{\omega{}^{\omega{}}}
b = \varepsilon_{\omega{}^{\omega{}^{\omega{}}}}
c = \varphi( \varepsilon_{\omega{}^{\omega{}^{\omega{}}}}, 0, 0)
ordCalc> opts both
ordCalc> list
a = ( w^( w^w ) )
a = \omega{}^{\omega{}^{\omega{}}}
b = epsilon( ( w^( w^w ) ))
b = \varepsilon_{\omega{}^{\omega{}^{\omega{}}}}
c = gamma( epsilon( ( w^( w^w ) )) )
c = \varphi( \varepsilon_{\omega{}^{\omega{}^{\omega{}}}}, 0, 0)
```

A.11.4 Member functions

The following demonstrates member functions.

```
ordCalc> a=psi(1,0,0,0,0)
Assigning psi( 1, 0, 0, 0, 0 ) to ‘a’.
ordCalc> a.listElts(3)

3 limitElements for psi( 1, 0, 0, 0, 0 )
le(1) = psi( 1, 0, 0, 0 )
le(2) = psi( psi( 1, 0, 0, 0 ) + 1, 0, 0, 0 )
le(3) = psi( psi( psi( 1, 0, 0, 0 ) + 1, 0, 0, 0 ) + 1, 0, 0, 0 )
End limitElements

Normal form:  psi( 1, 0, 0, 0, 0 )
ordCalc> b=a.limitElt(6)
Assigning psi( psi( psi( psi( psi( psi( 1, 0, 0, 0 ) + 1, 0, 0, 0 ) + 1, 0, 0,
0 ) + 1, 0, 0, 0 ) + 1, 0, 0, 0 ) + 1, 0, 0, 0 ) to ‘b’.
```

A.11.5 Veblen function of N ordinals

The following demonstrates Veblen functions of a finite number of ordinals.

The Veblen function with a finite number of parameters, psi(x1,x2,...xn) is built up from the function omega^x. psi(x) = omega^x. psi(1,x) enumerates the fixed points of omega^x. This is epsilon(x). Each additional variable diagonalizes the functions definable with existing variables. These functions can have any finite number of parameters.

```
ordCalc> a=psi(w,w)
Assigning psi( w, w ) to ‘a’.
ordCalc> b=psi(a,3,1)
Assigning psi( psi( w, w ), 3, 1 ) to ‘b’.
ordCalc> b.listElts(3)

3 limitElements for psi( psi( w, w ), 3, 1 )
le(1) = psi( psi( w, w ), 3, 0 ) + 1
le(2) = psi( psi( w, w ), 2, psi( psi( w, w ), 3, 0 ) + 1 )
le(3) = psi( psi( w, w ), 2, psi( psi( w, w ), 2, psi( psi( w, w ), 3, 0 ) + 1
) + 1 )
End limitElements

Normal form:  psi( psi( w, w ), 3, 1 )
ordCalc> c=psi(a,a,b,1,3)
Assigning psi( psi( w, w ), psi( w, w ), psi( psi( w, w ), 3, 1 ), 1, 3 ) to ‘c’.
ordCalc> c.listElts(3)

3 limitElements for psi( psi( w, w ), psi( w, w ), psi( psi( w, w ), 3, 1 ), 1,
3 )
le(1) = psi( psi( w, w ), psi( w, w ), psi( psi( w, w ), 3, 1 ), 1, 2 ) + 1
le(2) = psi( psi( w, w ), psi( w, w ), psi( psi( w, w ), 3, 1 ), 0, psi( psi(
w, w ), psi( w, w ), psi( psi( w, w ), 3, 1 ), 1, 2 ) + 1 )
le(3) = psi( psi( w, w ), psi( w, w ), psi( psi( w, w ), 3, 1 ), 0, psi( psi(
w, w ), psi( w, w ), psi( psi( w, w ), 3, 1 ), 0, psi( psi( w, w ), psi( w, w
), psi( psi( w, w ), 3, 1 ), 1, 2 ) + 1 ) + 1 )
End limitElements

Normal form:  psi( psi( w, w ), psi( w, w ), psi( psi( w, w ), 3, 1 ), 1, 3 )
```

A.11.6 Extended Veblen function

The following demonstrates Veblen functions iterated up to a recursive ordinal.

The extended Veblen function, psi_{a}(x1,x2,...,xn), iterates the idea of the Veblen function up to any recursive ordinal. The first parameter is the recursive ordinal of this iteration.

```
ordCalc> a=psi_{1}(1)
Assigning psi_{ 1}(1) to ‘a’.
ordCalc> a.listElts(4)
```

```
4 limitElements for psi_{ 1}(1)
le(1) = ( w^( psi_{ 1} + 1 ) )
le(2) = psi( psi_{ 1} + 1, 0 )
le(3) = gamma( psi_{ 1} + 1 )
le(4) = psi( psi_{ 1} + 1, 0, 0, 0 )
End limitElements

Normal form:  psi_{ 1}(1)
ordCalc> b=psi_{w+1}(3)
Assigning psi_{ w + 1}(3) to 'b'.
ordCalc> b.listElts(4)

4 limitElements for psi_{ w + 1}(3)
le(1) = psi_{ w}(psi_{ w + 1}(2) + 1)
le(2) = psi_{ w}(psi_{ w + 1}(2) + 1, 0)
le(3) = psi_{ w}(psi_{ w + 1}(2) + 1, 0, 0)
le(4) = psi_{ w}(psi_{ w + 1}(2) + 1, 0, 0, 0)
End limitElements

Normal form:  psi_{ w + 1}(3)
```

A.11.7 Admissible countable ordinal notations

The following demonstrates countable admissible level ordinal notations.

Countable admissible level notations, omega_{k,g}(x1,x3,..,xn) extend the idea of recursive notation to larger countable ordinals. The first parameter is the admissible level. omega_{1} is the Church-Kleene ordinal. The remaining parameters are similar to those defined for smaller ordinal notations.

```
ordCalc> a=w_{1}(1)
Assigning omega_{ 1}(1) to 'a'.
ordCalc> a.listElts(4)

4 limitElements for omega_{ 1}(1)
le(1) = psi_{ omega_{ 1} + 1}
le(2) = psi_{ psi_{ omega_{ 1} + 1} + 1}
le(3) = psi_{ psi_{ psi_{ omega_{ 1} + 1} + 1} + 1}
le(4) = psi_{ psi_{ psi_{ psi_{ omega_{ 1} + 1} + 1} + 1} + 1}
End limitElements

Normal form:  omega_{ 1}(1)
```

A.11.8 Admissible notations drop down parameter

The following demonstrates admissible notations dropping down one level.

Admissible level ordinals have the a limit sequence defined in terms of lower levels. The lowest admissible level is that of recursive ordinals. To implement this definition of limit sequence, a trailing parameter in square brackets is used. This parameter (if present) defines an ordinal at one admissible level lower than indicated by other parameters.

```
ordCalc> a=w_{1}[1]
Assigning omega_{ 1}[ 1] to 'a'.
ordCalc> a.listElts(4)

4 limitElements for omega_{ 1}[ 1]
le(1) = w
le(2) = psi_{ w}
le(3) = psi_{ psi_{ w} + 1}
le(4) = psi_{ psi_{ psi_{ w} + 1} + 1}
End limitElements

Normal form:  omega_{ 1}[ 1]
ordCalc> b=w_{1}
Assigning omega_{ 1} to 'b'.
ordCalc> c=b.limitOrd(w^3)
Assigning omega_{ 1}[ ( w^3 )] to 'c'.
ordCalc> c.listElts(4)

4 limitElements for omega_{ 1}[ ( w^3 )]
le(1) = omega_{ 1}[ ( w^2 )]
le(2) = omega_{ 1}[ (( w^2 )*2 )]
le(3) = omega_{ 1}[ (( w^2 )*3 )]
le(4) = omega_{ 1}[ (( w^2 )*4 )]
End limitElements

Normal form:  omega_{ 1}[ ( w^3 )]
ordCalc> d=w_{5,c}(3,0)
Assigning omega_{ 5, omega_{ 1}[ ( w^3 )]}(3, 0) to 'd'.
ordCalc> d.listElts(4)

4 limitElements for omega_{ 5, omega_{ 1}[ ( w^3 )]}(3, 0)
le(1) = omega_{ 5, omega_{ 1}[ ( w^3 )]}(2, 1)
le(2) = omega_{ 5, omega_{ 1}[ ( w^3 )]}(2, omega_{ 5, omega_{ 1}[ ( w^3 )]}(2,
1) + 1)
le(3) = omega_{ 5, omega_{ 1}[ ( w^3 )]}(2, omega_{ 5, omega_{ 1}[ ( w^3 )]}(2,
omega_{ 5, omega_{ 1}[ ( w^3 )]}(2, 1) + 1) + 1)
```

```
le(4) = omega_{ 5, omega_{ 1}[ ( w^3 )]}(2, omega_{ 5, omega_{ 1}[ ( w^3 )]}(2,
omega_{ 5, omega_{ 1}[ ( w^3 )]}(2, omega_{ 5, omega_{ 1}[ ( w^3 )]}(2, 1) + 1)
+ 1) + 1)
End limitElements

Normal form:  omega_{ 5, omega_{ 1}[ ( w^3 )]}(3, 0)
```

A.11.9 Admissible notations context parameter

The following demonstrates admissible ordinal context parameters.

The context parameter in admissible level ordinals allows one to use any notation at any admissible level to define notations at any lower admissible level or to define recursive ordinals.

```
ordCalc> a=[[1]]w_{1}
Assigning [[1]]omega_{ 1} to ‘a’.
ordCalc> a.listElts(4)

4 limitElements for [[1]]omega_{ 1}
le(1) = omega_{ 1}[ w]
le(2) = omega_{ 1}[ omega_{ 1}[ w]]
le(3) = omega_{ 1}[ omega_{ 1}[ omega_{ 1}[ w]]]
le(4) = omega_{ 1}[ omega_{ 1}[ omega_{ 1}[ omega_{ 1}[ w]]]]
End limitElements

Normal form:  [[1]]omega_{ 1}
```

A.11.10 Lists of ordinals

The following shows how lists work.

Lists are a sequence of ordinals (including integers). A list can be assigned to a variable just as a single ordinal can be. In most circumstances lists are evaluated as the first ordinal in the list. In ‘limitOrdLst’ all of the list entries are used. These member functions return a list with an input list

```
ordCalc> lst = 1, 12, w, gamma(w^w), w1
Assigning 1, 12, w, gamma( ( w^w ) ), omega_{ 1} to ‘lst’.
ordCalc> a=w1.limitOrdLst(lst)
( omega_{ 1} ).limitOrd( 12 ) = omega_{ 1}[ 12]
( omega_{ 1} ).limitOrd( w ) = omega_{ 1}[ w]
( omega_{ 1} ).limitOrd( gamma( ( w^w ) ) ) = omega_{ 1}[ gamma( ( w^w ) )]
Assigning omega_{ 1}[ 12], omega_{ 1}[ w], omega_{ 1}[ gamma( ( w^w ) )] to ‘a’.
ordCalc> bg = w_{w+33}
```

```
Assigning omega_{ w + 33} to ‘bg’.
ordCalc> c=bg.limitOrdLst(lst)
( omega_{ w + 33} ).limitOrd( 12 ) = omega_{ w + 33}[ 12]
( omega_{ w + 33} ).limitOrd( w ) = omega_{ w + 33}[ w]
( omega_{ w + 33} ).limitOrd( gamma( ( w^w ) ) ) = omega_{ w + 33}[ gamma( ( w^w
) )]
( omega_{ w + 33} ).limitOrd( omega_{ 1} ) = omega_{ w + 33}[ omega_{ 1}]
Assigning omega_{ w + 33}[ 12], omega_{ w + 33}[ w], omega_{ w + 33}[ gamma( (
w^w ) )], omega_{ w + 33}[ omega_{ 1}] to ‘c’.
```

A.11.11 List of descending trees

The following shows how to construct a list of descending trees.

‘desLimitOrdLst’ iterates ’limitOrdLst’ to a specified ‘depth’. The first parameter is the integer depth of iteration and the second is the list of parameters to be used. This routine first takes‘limitOrd’ of each element in the second parameter creating a list of outputs. It then takes this list and evaluates ‘limitOrd’ for each of these values at each entry in the original parameter list. All of these results are combined in a new list and the process is iterated ‘depth’ times. The number of results grows exponentially with ‘depth’.

```
ordCalc> lst = 1, 5, w, psi(2,3)
Assigning 1, 5, w, psi( 2, 3 ) to ‘lst’.
ordCalc> bg = w_{3}
Assigning omega_{ 3} to ‘bg’.
ordCalc> d= bg.desLimitOrdLst(2,lst)
( omega_{ 3} ).limitOrd( 1 ) = omega_{ 3}[ 1]
( omega_{ 3} ).limitOrd( 5 ) = omega_{ 3}[ 5]
( omega_{ 3} ).limitOrd( w ) = omega_{ 3}[ w]
( omega_{ 3} ).limitOrd( psi( 2, 3 ) ) = omega_{ 3}[ psi( 2, 3 )]
Descending to 1 for omega_{ 3}
( omega_{ 3}[ 1] ).limitOrd( 1 ) = omega_{ 2}
( omega_{ 3}[ 1] ).limitOrd( 5 ) = omega_{ 2, omega_{ 2, omega_{ 2, omega_{ 2,
omega_{ 2} + 1} + 1} + 1} + 1}
( omega_{ 3}[ 5] ).limitOrd( 1 ) = omega_{ 3}[ 4]
( omega_{ 3}[ 5] ).limitOrd( 5 ) = omega_{ 2, omega_{ 2, omega_{ 2, omega_{ 2,
omega_{ 3}[ 4] + 1} + 1} + 1} + 1}
( omega_{ 3}[ w] ).limitOrd( 1 ) = omega_{ 3}[ 1]
( omega_{ 3}[ w] ).limitOrd( 5 ) = omega_{ 3}[ 5]
( omega_{ 3}[ psi( 2, 3 )] ).limitOrd( 1 ) = omega_{ 3}[ psi( 2, 2 ) + 1]
( omega_{ 3}[ psi( 2, 3 )] ).limitOrd( 5 ) = omega_{ 3}[ epsilon( epsilon( epsilon(
epsilon( psi( 2, 2 ) + 1) + 1) + 1) + 1)]
Assigning omega_{ 3}[ 1], omega_{ 3}[ 5], omega_{ 3}[ w], omega_{ 3}[ psi( 2, 3
)], omega_{ 2}, omega_{ 2, omega_{ 2, omega_{ 2, omega_{ 2, omega_{ 2} + 1} + 1}
+ 1} + 1}, omega_{ 3}[ 4], omega_{ 2, omega_{ 2, omega_{ 2, omega_{ 2, omega_{ 3}[
4] + 1} + 1} + 1} + 1}, omega_{ 3}[ 1], omega_{ 3}[ 5], omega_{ 3}[ psi( 2, 2 )
+ 1], omega_{ 3}[ epsilon( epsilon( epsilon( epsilon( psi( 2, 2 ) + 1) + 1) +
1) + 1)] to ‘d’.
```

References

[1] W. Buchholz. A new system of proof-theoretic ordinal functions. *Ann. Pure Appl. Logic*, 32:195–207, 1986. 42

[2] Paul Budnik. *What is and what will be: Integrating spirituality and science.* Mountain Math Software, Los Gatos, CA, 2006. 4, 6, 64, 68

[3] Paul Budnik. What is Mathematics About? In Paul Ernest, Brian Greer, and Bharath Sriraman, editors, *Critical Issues in Mathematics Education*, pages 283–292. Information Age Publishing, Charlotte, North Carolina, 2009. 4, 64, 68

[4] Paul J. Cohen. *Set Theory and the Continuum Hypothesis.* W. A. Benjamin Inc., New York, Amsterdam, 1966. 4, 6, 68

[5] Solomon Feferman. Does mathematics need new axioms? *American Mathematical Monthly*, 106:99–111, 1999. 67

[6] Jean H. Gallier. What's so Special About Kruskal's Theorem And The Ordinal Γ_0? A Survey Of Some Results In Proof Theory. *Annals of Pure and Applied Logic*, 53(2):199–260, 1991. 7, 17, 19, 68

[7] Stephen Hawking. *God Created the Integers: The Mathematical Breakthroughs that Changed History.* Running Press, Philidelphia, PA, 2005. 9

[8] W. A. Howard. A system of abstract constructive ordinals. *The Journal of Symbolic Logic*, 37(2):355–374, 1972. 44, 68

[9] G. Kreisel and Gerald E. Sacks. Metarecursive sets. *The Journal of Symbolic Logic*, 30(3):318–338, 1965. 8

[10] Benoît Mandelbrot. Fractal aspects of the iteration of $z \mapsto \lambda z(1-z)$ for complex λ and z. *Annals of the New York Academy of Sciences*, 357:49–259, 1980. 41

[11] Larry W. Miller. Normal functions and constructive ordinal notations. *The Journal of Symbolic Logic*, 41(2):439–459, 1976. 7, 17, 19, 68

[12] Michael Rathjen. How to develop Proof-Theoretic Ordinal Functions on the basis of admissible ordinals. *Mathematical Logic Quarterly*, 39:47–54, 2006. 42

[13] Hartley Rogers Jr. *Theory of Recursive Functions and Effective Computability.* McGraw Hill, New York, 1967. 8

[14] Gerald Sacks. *Higher recursion theory.* Springer Verlag, New York, 1990. 39

[15] Oswald Veblen. Continuous increasing functions of finite and transfinite ordinals. *Transactions of the American Mathematical Society*, 9(3):280–292, 1908. 7, 17, 68

[16] A. N. Whitehead and Russell Bertrand. *Principia Mathematica*, volume 1-3. Cambridge University Press, 1918. 40

Index

The defining reference for a phrase, if it exists, has the page *number* in *italics*.

This index is semiautomated with multiple entries created for some phrases and subitems automatically detected. Hand editing would improve things, but is not practical for a manual describing software and underlying theory both of which are evolving.

Formatted: January 17, 2010