A Computational Approach to the Ordinal Numbers

Documents ordCalc_0.1

Paul Budnik
Mountain Math Software
paul@mtnmath.com

Contents

1 Intended audience **2**

2 Introduction **3**

3 Ordinal notations **4**
3.1 Ordinal functions and fixed points 5

4 Program structure **6**
4.1 `virtual` functions and subclasses 7
4.2 Ordinal normal forms 7
4.3 Memory management 8

5 `Ordinal` base `class` **8**
5.1 `Ordinal compare` member function 9
5.2 `Ordinal limitElement` member function 9
5.3 `Ordinal` operators 10
5.3.1 `Ordinal` addition 11
5.3.2 `Ordinal` multiplication 12
5.3.3 `Ordinal` exponentiation 13

6 The Veblen hierarchy and beyond **17**
6.1 The delta operator 17
6.2 A finite function hierarchy 18
6.3 The finite function normal form 19
6.4 `limitElement` for finite functions 19
6.5 An iterative function hierarchy 21

7 `FiniteFuncOrdinal` class **21**
7.1 `FiniteFuncOrdinal compare` member function 22
7.2 `FiniteFuncOrdinal limitElement` member function 24
7.3 `FiniteFuncOrdinal fixedPoint` member function 27
7.4 `FiniteFuncOrdinal` operators 27
7.4.1 `FiniteFuncOrdinal` multiplication 27
7.4.2 `FiniteFuncOrdinal` exponentiation 30

8 `IterFuncOrdinal` class **32**
8.1 `IterFuncOrdinal compare` member function 32
8.2 `IterFuncOrdinal limitElement` member function 34
8.3 `IterFuncOrdinal fixedPoint` member function 36
8.4 `IterFuncOrdinal` operators 36

9 Philosophical Issues **39**

A Command line interface **41**
A.1 Introduction 41
A.2 Ordinals 41
A.3 Syntax 42
A.4 Commands 42
A.4.1 All commands 42
A.4.2 Commands with options 43
A.5 Member functions 43
A.6 Comparison operators 44
A.7 Examples 44
A.7.1 Simple ordinal arithmetic 44
A.7.2 Comparison operators 44
A.7.3 Display options 45
A.7.4 Member functions 45

References **47**

1 Intended audience

The ordinal calculator documented here is a tool for learning about the ordinal hierarchy and ordinal notations. It is also intended as a research tool. Appendix A (available as a separate manual, "Using the ordinal calculator") is the user's manual. It should be all you need download and use the program.

This document is intended for those who want to understand the structure of the program and/or the theory on which it is based. It is hoped that some may

want to modify or extend the program. All the source code and documentation (including this manual) is licensed for that purpose under the GNU General Public License, version 2.

This paper 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.

2 Introduction

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.

[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. All larger ordinals are constructed by taking the union of *all* smaller ones. Ordinals are of three types. 0 or the empty set is the smallest ordinal and the only ordinal 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.

A limit ordinal must contain *all* smaller ordinals. The same is true of successor ordinals. Thus 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 $a = b$. Taking the successor of an ordinal is the definition of the operation $+1$ in set theory.

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.

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 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].

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 larger ordinals but an incomplete variant of it can be defined.

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) 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.

3 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 ${\omega_1}$[4]. 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:

[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.

[4] 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 we can simplify the notation for ω_1^{CK} to ω_1.

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 $a = b$[5]. 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 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[6]. 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 is crucial to expanding the ordinal hierarchy. Beyond a certain point this is not possible without using computers as a research tool.

3.1 Ordinal functions and fixed points

Notations for ordinals are usually defined with strictly increasing ordinal functions on the countable ordinals with $f(0) > 0$. A fixed point of a function f is an ordinal a with $f(a) = a$. For example a fixed point of $f(x) = n + x (n < \omega)$ is ω or any

[5] 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.

[6] 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 tree of the successor of b or $b + 1$.

3. If it is tree for a sequence of ordinals c_i such that $\bigcup_i c_i = c$ and $c \leq a$, then it is true of c.

limit ordinal. 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[6, 5, 4]. The first of these functions, $\varphi(1, x)$ enumerates the fixed points of ω^x. $\varphi(1, x) = \epsilon_x$.

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 5 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. In addition starting with a finite integer and adding ω to it cannot get past ω. $n + \omega = \omega$ for all finite integers n.[7] The first fixed point for the function, ω^x, is the ordinal $\epsilon_0 = \omega + \omega^\omega + \omega^{(\omega^\omega)} + \omega^{(\omega^{(\omega^\omega)})} + ...$ (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 is the union of all the 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.

4 Program structure

The C++ programming language[8] 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

[7] 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.

[8] 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 nonstatic member functions can only be made from an instance of the `class`. Data `class` members in the function definition refer to the particular instance of the `class` from which the function is called.

any subclass of `Ordinal` to be passed to it as an argument. The reverse does not hold.

4.1 `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 version 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`.

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.

4.2 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:

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

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

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

Because $\epsilon_0 = \omega^{\epsilon_0}$ 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 6) defines ϵ_0 as $\varphi(1,0)$. Thus the normal form for ϵ_0 would be $\varphi(1,0)$ and not $\omega^{\varphi(1,0)}$.

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`.

4.3 Memory management

Most infinite `Ordinals` are constructed from previously defined `Ordinal`s. Those constructions need to reference the ordinals used in defining them. The `Ordinal`s used in the definition of another `Ordinal` must not be deleted while the object that uses them still exists. This often requires that `Ordinal`s 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 `Ordinal`s using `new` will eventually exhaust available memory.

5 `Ordinal` base `class`

The finite (integer) `Ordinal`s and ω, the ordinal that contains all integers, are defined using the constructor[9] for `class Ordinal`.[10]. Other base `class` ordinals are usually defined using addition, multiplication and exponentiation (see Section 5.3) in expressions that can include integer `Ordinal`s and ω (C++ name `omega`).

To define a finite `Ordinal`, such as 12, one can write the code "`const Ordinal& twelve = * new Ordinal(12)`". The `Ordinal`s for `zero`, `one` and `omega` are defined in global name space `ord`[11] The variables `two` through `six` are defined as members of `class Ordinal`[12].

The standard operations for ordinal arithmetic (+, * for $\times$ and ^ for exponentiation) are defined for all `Ordinal` instances. Expressions involving exponentiation

[9]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.

[10]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.

[11]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.

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

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

Table 1: `Ordinal` C++ code examples

must use parenthesis to indicate precedence because C++ gives lower precedence to `^` then it does to addition and multiplication[13]. In C++ the standard use of `^` is for the boolean operation exclusive or. Some examples of infinite `Ordinal`s created by `Ordinal` expressions are shown in Table 1.

5.1 `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 `Ordinal`s (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 `Ordinal`s) 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.

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.

5.2 `Ordinal limitElement` member function

`Ordinal` member function `limitElement` has a single integer parameter. 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 5.

In the following description mathematical notation is mixed with C++ code.

[13]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)`
$\alpha_k = 0 \wedge i \leq n_k$	$\gamma + i$
$\alpha_k = 0 \wedge i > n_k$	$\gamma + n_k$
$\alpha_k = 1$	$\gamma + i$
$\alpha_k > 1 \wedge \alpha_k$ 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
$\omega \times 8$	$\omega \times 7 + 1$	$\omega \times 7 + 2$	$\omega \times 7 + 10$	$\omega \times 7 + 100$	$\omega \times 7 + 786$
ω^2	ω	$\omega \times 2$	$\omega \times 10$	$\omega \times 100$	$\omega \times 786$
ω^3	ω^2	$\omega^2 \times 2$	$\omega^2 \times 10$	$\omega^2 \times 100$	$\omega^2 \times 786$
ω^ω	ω	ω^2	ω^{10}	ω^{100}	ω^{786}
$\omega^{\omega+2}$	$\omega^{\omega+1}$	$\omega^{\omega+1} \times 2$	$\omega^{\omega+1} \times 10$	$\omega^{\omega+1} \times 100$	$\omega^{\omega+1} \times 786$
$\omega^{\omega^{\omega}}$	$\omega^{\omega^{\omega}}$	$\omega^{\omega^{\omega^2}}$	$\omega^{\omega^{\omega^{10}}}$	$\omega^{\omega^{\omega^{100}}}$	$\omega^{\omega^{\omega^{786}}}$

Table 3: `Ordinal::limitElement` examples.

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. `limitElement` operates mainly on the last or least significant term of the normal form expression for an ordinal notation. 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 $\omega^{\alpha_k}(n_k - 1)$. All outputs from `limitElement` include γ plus a final term that varies according to several conditions as shown in Table 2. Table 3 gives some examples.

5.3 `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 $\wedge$[14] Arithmetic on the ordinals

[14]'$\wedge$' 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

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
$(\omega \times 4 + 12) \times \omega$	ω^2	`(omega*4+12)*omega`
$\omega \times (\omega \times 4 + 12)$	$\omega^2 \times 4 + \omega \times 12$	`omega*(omega*4+12)`
$\omega^{\omega \times (\omega+3)}$	$\omega^{\omega^2+\omega \times 3}$	`omega^(omega*(omega+3))`
$(\omega + 4) \times (\omega + 5)$	$\omega^2 + \omega \times 5 + 4$	`(omega+4)*(omega+5)`
$(\omega + 2)^{(\omega+2)}$	$\omega^{\omega+2} + \omega^{\omega+1} \times 2 + \omega^{\omega} \times 2$	`(omega+2)^(omega+2)`
$(\omega + 3)^{(\omega+3)}$	$\omega^{\omega+3} + \omega^{\omega+2} \times 3 +$ $\omega^{\omega+1} \times 3 + \omega^{\omega} \times 3$	`(omega+3)^(omega+3)`

Table 5: Ordinal arithmetic examples

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.

5.3.1 `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:

$$\alpha, \beta \; ordinals \; \wedge \alpha \geq \beta \rightarrow \beta + \alpha * \omega = \alpha * \omega \tag{2}$$

`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 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.

`(x^y)+1`.

5.3.2 `Ordinal` multiplication

Multiplication of infinite ordinals is complicated by the way addition works. For example, from Equation 2:

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

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. (Addition is organized similarly with a member function of `CantorNormalElement` that adds terms and a separate procedure that scans the terms of an ordinal.) When a subclass of `Ordinal` is defined one also needs to define a subclass of `CantorNormalElement`.

Each subclass of `CantorNormalElement` is assigned a `codeLevel`. These integers increase for each subclass[15]. The value for the base class is `cantorCodeLevel`. Any 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

[15] One cannot determine the relative size of two `CantorNormalElement`s with unequal `codeLevel`s by comparing `codeLevel`s. This is possible for the first few subclasses, but a later subclass defines the ordinal of the recursive ordinal (ω_1^{CK}) and beyond. From that point on new subclasses are not continuous in the ordinals they define and they partially fill gaps at multiple levels in the ordinal hierarchy.

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.

5.3.3 `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[16]. If the base is finite and the exponent is infinite, the product of the infinite terms in the exponent is 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.

[16]Be careful with integer exponentiation. A large exponent can require a great deal of memory to store the result. This can cause the computer to slow dramatically and ultimately kill the program.

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

Table 6: `Ordinal` multiply examples

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

Table 7: C++ code for ordinal exponential

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}
$\omega + 1$	$\omega^3 \times 2 + 2$	$\omega^{\omega^3\times2+2} + \omega^{\omega^3\times2+1} + \omega^{\omega^3\times2}$
$\omega + 1$	$\omega^4 + 3$	$\omega^{\omega^4+3} + \omega^{\omega^4+2} + \omega^{\omega^4+1} + \omega^{\omega^4}$
$\omega + 1$	$\omega^\omega \times 3$	$\omega^{\omega^\omega\times3}$
$\omega + 2$	$\omega + 1$	$\omega^{\omega+1} + \omega^\omega \times 2$
$\omega + 2$	$\omega + 2$	$\omega^{\omega+2} + \omega^{\omega+1} \times 2 + \omega^\omega \times 2$
$\omega + 2$	ω^3	ω^{ω^3}
$\omega + 2$	$\omega^3 \times 2 + 2$	$\omega^{\omega^3\times2+2} + \omega^{\omega^3\times2+1} \times 2 + \omega^{\omega^3\times2} \times 2$
$\omega + 2$	$\omega^4 + 3$	$\omega^{\omega^4+3} + \omega^{\omega^4+2} \times 2 + \omega^{\omega^4+1} \times 2 + \omega^{\omega^4} \times 2$
$\omega + 2$	$\omega^\omega \times 3$	$\omega^{\omega^\omega\times3}$
ω^3	$\omega + 1$	$\omega^{\omega+3}$
ω^3	$\omega + 2$	$\omega^{\omega+6}$
ω^3	ω^3	ω^{ω^3}
ω^3	$\omega^3 \times 2 + 2$	$\omega^{\omega^3\times2+6}$
ω^3	$\omega^4 + 3$	ω^{ω^4+9}
ω^3	$\omega^\omega \times 3$	$\omega^{\omega^\omega\times3}$
$\omega^3 \times 2 + 2$	$\omega + 1$	$\omega^{\omega+3} \times 2 + \omega^\omega \times 2$
$\omega^3 \times 2 + 2$	$\omega + 2$	$\omega^{\omega+6} \times 2 + \omega^{\omega+3} \times 4 + \omega^\omega \times 2$
$\omega^3 \times 2 + 2$	ω^3	ω^{ω^3}
$\omega^3 \times 2 + 2$	$\omega^3 \times 2 + 2$	$\omega^{\omega^3\times2+6} \times 2 + \omega^{\omega^3\times2+3} \times 4 + \omega^{\omega^3\times2} \times 2$
$\omega^3 \times 2 + 2$	$\omega^4 + 3$	$\omega^{\omega^4+9} \times 2 + \omega^{\omega^4+6} \times 4 + \omega^{\omega^4+3} \times 4 + \omega^{\omega^4} \times 2$
$\omega^3 \times 2 + 2$	$\omega^\omega \times 3$	$\omega^{\omega^\omega\times3}$
$\omega^4 + 3$	$\omega + 1$	$\omega^{\omega+4} + \omega^\omega \times 3$
$\omega^4 + 3$	$\omega + 2$	$\omega^{\omega+8} + \omega^{\omega+4} \times 3 + \omega^\omega \times 3$
$\omega^4 + 3$	ω^3	ω^{ω^3}
$\omega^4 + 3$	$\omega^3 \times 2 + 2$	$\omega^{\omega^3\times2+8} + \omega^{\omega^3\times2+4} \times 3 + \omega^{\omega^3\times2} \times 3$
$\omega^4 + 3$	$\omega^4 + 3$	$\omega^{\omega^4+12} + \omega^{\omega^4+8} \times 3 + \omega^{\omega^4+4} \times 3 + \omega^{\omega^4} \times 3$
$\omega^4 + 3$	$\omega^\omega \times 3$	$\omega^{\omega^\omega\times3}$
$\omega^\omega \times 3$	$\omega + 1$	$\omega^{\omega^2+\omega} \times 3$
$\omega^\omega \times 3$	$\omega + 2$	$\omega^{\omega^2+\omega\times2} \times 3$
$\omega^\omega \times 3$	ω^3	ω^{ω^4}
$\omega^\omega \times 3$	$\omega^3 \times 2 + 2$	$\omega^{\omega^4\times2+\omega\times2} \times 3$
$\omega^\omega \times 3$	$\omega^4 + 3$	$\omega^{\omega^5+\omega\times3} \times 3$
$\omega^\omega \times 3$	$\omega^\omega \times 3$	$\omega^{\omega^\omega\times3}$

Table 8: `Ordinal` exponential examples

6 The Veblen hierarchy and beyond

This section gives a brief overview of the Veblen hierarchy and the Δ operator as developed in set theory. See [6, 5, 4] for a more a complete treatment. This is followed by the development of a computational approach for constructing notations for these ordinals and beyond.

The Veblen hierarchy extends the recursive ordinals beyond ϵ_0. A Veblen hierarchy can be constructed from any strictly increasing continuous function[17] f, whose domain and range are the countable ordinals such that $f(0) > 0$. $f(x) = \omega^x$ satisfies these conditions and is the usual starting point for building Veblen hierarchies. The idea of the hierarchy is to define a new function from an existing one such that the new function enumerates the fixed points of the previous one. A fixed point of a function 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.

6.1 The delta operator

The Δ operator is defined as follows[5, 4].

- $\Delta_0(\psi)$ 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 $\Delta_\gamma(\varphi)$.
- $\varphi_0^\alpha = \varphi$.

[17] 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.

- $\varphi^{\alpha}_{\beta'} = \Delta_{\alpha}(\varphi^{\alpha}_{\beta})$.
- φ^{α}_{β} for β a limit ordinal enumerates $\bigcap_{\gamma<\beta}$ range $\varphi^{\alpha}_{\gamma}$.

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 17) 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.

6.2 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[18] 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.

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

[18] 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.

6.3 The finite function normal form

φ is constructed bottom up. Each higher level function enumerates "computational fixed points" of the next lowest level function. Computational fixed points are the unions of infinite sequences of notations with a limit that is not reachable by finite unions of lower level notations. These are defined by the `Ordinal::limitElement` member function introduced in Section 5.2. To define them for this extended ordinal hierarchy (beyond ϵ_0) requires an expanded normal form shown in Equation 6 which is an extension of Equation 1.

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

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

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

α_i and $\beta_{j,i}$ are ordinals and n_k are integers > 0.

Note that $\varphi(\beta) = \omega^\beta$ so the above includes the Cantor normal form terms. To eliminate fixed points (for example $\varphi(1,0) = \epsilon_0 = \omega^{\epsilon_0} = \varphi(\varphi(1,0))$) the rule is adopted that all normal forms must be reduced to the simplest expression that can represent them. To this end an algorithm for detecting fixed points is used in creating normal forms (see Section 7.3). Fixed point detection depends on the partial definition of φ which comes from subroutine `limitElement`. Each ordinal represented by φ is defined by the enumeration of a possibly infinite sequence of smaller ordinal notations specified in the `limitElement` routine.

6.4 `limitElement` for finite functions

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 $\geq \epsilon_0$) 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.

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

$\alpha = \varphi(\beta_1, \beta_2, ..., \beta_m)$ 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 7.3 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());`

X This column gives the exit code from `limitElement` (see Section 7.2).
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)`

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.

6.5 An iterative function hierarchy

Recall that a finite functional hierarchy at each level, iterates a function at the next lowest level up to any `Ordinal`[19] The base function of the hierarchy is ω^x.

To generalize this one would like to iterate the definition of a function hierarchy not just any finite number of times but up to any `Ordinal`. The key to defining this iteration is the `limitElement` member function which enumerates notations the union of which define the ordinal a new notation stands for. To support this expanded notation we extend the normal form in Equation 6.

$$\alpha = \alpha_1 n_1 + \alpha_2 n_2 + \alpha_3 n_3 + ... + \alpha_k n_k \tag{7}$$

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

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

α_i, γ_i and $\beta_{i.j}$ are ordinals and n_k are integers > 0.

γ_i, the subscript to φ, is the `Ordinal` the function hierarchy is iterated up to. $\varphi_0(\beta_1, \beta_2, ..., \beta_m) = \varphi(\beta_1, \beta_2, ..., \beta_n)$. $\varphi_1(0)$ is the notation for an infinite union of `FiniteFuncOrdinal`s. Specifically it represents the union of ordinals with notations: $\varphi(1), \varphi(1,0).\varphi(1,0,0), \varphi(1,0,0,0), ...$ Note $\varphi(1) = \omega, \varphi(1,0) = \epsilon_0$ and $\varphi(1,0,0) = \Gamma_0$. The β_i index the iteration of functions that operate on the base function defined by $\varphi_\gamma(0)$.

`limitElement` for this hierarchy is shown in Table 10. This is an extension of Table 9.

7 `FiniteFuncOrdinal class`

The `FiniteFuncOrdinal class` is derived from the `Ordinal` base class. It implements the ordinal notations, using $\varphi(\alpha_1\alpha_2, ..., \alpha_n)$, in terms of the normal form given by Equation 6. Each term of this normal form, each $\alpha_i n_i$, is represented by an instance of `class FiniteFuncNormalElement` or class `CantorNormalForm` from which `FiniteFuncNormalElement` is derived.

[19]Upper case 'ordinal' in `tty font` (`Ordinal`) refers to the expandable C++ class of ordinal notations in this implementation.

$\alpha = \varphi_\gamma(\beta_1, \beta_2, ..., \beta_m)$ 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 8.2).
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)`

The `FiniteFuncOrdinal class` should not be used directly to create ordinal notations. Instead use functions `psi` or `finiteFunctional` to create an instance of `FiniteFuncOrdinal`. `psi` construct notations for the initial Veblen hierarchy. It requires two parameters. (For the single parameter case, $\varphi(\alpha) = \omega^\alpha$, the `Ordinal` base class or the exponentiation operator, $\wedge$, should be used.) `finiteFunctional` is for Γ_0[20]and larger ordinals. Some examples are show in Table 11. The 'Ordinal' column of this table is created using `Ordinal::texNormalForm` which uses the standard notation for ϵ_x and Γ_x 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 the table is an example of this.

`FiniteFuncOrdinal` can be called with 3 or 4 parameters. For additional parameters, it can be called with an array of pointers to ordinal notations. The last entry in the array must be `NULL`. `createParameters` can be used to create this array as shown in the last entry in the table. `createParameters` can have 1 to 9 parameters all of which must be pointers to `Ordinals`[21].

7.1 `FiniteFuncOrdinal compare` member function

`FiniteFuncOrdinal::compare` supports the extended normal form in Equation 6. The $\beta_{j,i}$ in that equation are represented by the elements of array `funcParameters`

[20] Γ_0 is the smallest ordinal not accessible from the first Veblen hierarchy. It is $\varphi(1, 0, 0)$ in the notation used here It is called the Feferman-Schütte ordinal. ($\Gamma_\alpha = \varphi(1, \alpha, 0)$.

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

C++ code	Ordinal
`psi(zero,omega)`	ω^ω
`psi(one,zero)`	ϵ_0
`psi(one,one)`	ϵ_1
`psi(eps0,one)`	$\varphi(\epsilon_0, 1)$
`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)`	$\varphi(1, \epsilon_0, \omega)$
`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: `FiniteFuncOrdinal` C++ code examples

in the C++ program. The `compare virtual` functions overridden by `class`es `FiniteFuncOrdinal` and `FiniteFuncNormalElement` are those that compare terms of a normal form, `CantorNormalElement`s, and a single term against a full ordinal notation, `class OrdinalImpl`[22]. The latter `compare` function is trivial except for its dependence on the former.

`FiniteFuncNormalElement::compare` with a single `CantorNormalElement` argument is the main function that implements `compare`. It overrides `CantorNormalElement::compare` with the same argument (see Section 5.1). It outputs 1, 0 or -1 if the object it is called from is greater than equal to or less than its argument term.

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`[23] 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 of `Ordinal` parameters are used to construct a `FiniteFuncNormalElement`. Thus a series of tests is required. This is facili-

[22] `OrdinalImpl` is defined by `class Ordinal` for internal use only.

[23] `finiteFuncCodeLevel` is the `codeLevel`(see Section 5.3.2) of a `FiniteFuncNormalElement`.

tated by a member function `CantorNormalElement::getMaxParameter` that returns the largest parameter used in constructing this normal form term[24]. 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 $\texttt{trm} \geq \omega^{\texttt{trm.getMaxParameter}()}$ and this will swamp the effect of any finite `factor`.

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

1. If `trm.codeLevel` < `finiteFuncCodeLevel` the first (largest) term of the exponent of the argument is compared to `this`, ignoring the two `factor`s. If the result is nonzero that result 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.

If the above is not decisive `FiniteFuncNormalElement::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 this routine returns that the parameters of the two terms are identical the difference in the `factor`s of the two terms is returned.

7.2 `FiniteFuncOrdinal limitElement` member function

`FiniteFuncOrdinal::limitElement` overrides `Ordinal::limitElement` described in Section 5.2. 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 `FiniteFuncOrdinal class` instance `limitElement` is called from. This will be referred to as the input `Ordinal` to `limitElement`.

As with `Ordinal::limitElement`, all but the last term of the normal form of the result are copied 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 `Ordinal`.

[24]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.

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 12.

FiniteFuncOrdinal	limitElement		
ω	1	2	4
ϵ_0	ω	ω^ω	$\omega^{\omega^{\omega^\omega}}$
$\varphi(2,0)$	ϵ_1	ϵ_{ϵ_1+1}	$\epsilon_{\epsilon_{\epsilon_{\epsilon_1+1}+1}+1}$
$\varphi(\omega,\omega)$	$\varphi(\omega,1)$	$\varphi(\omega,2)$	$\varphi(\omega,4)$
Γ_0	ϵ_0	$\varphi(\epsilon_0+1,0)$	$\varphi(\varphi(\varphi(\epsilon_0+1,0)+1,0)+1,0)$
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	$\varphi(2,0)$	$\varphi(4,0)$
$\varphi(\omega,1)$	$\epsilon_{\varphi(\omega,0)+1}$	$\varphi(2,\varphi(\omega,0)+1)$	$\varphi(4,\varphi(\omega,0)+1)$
$\varphi(\omega,0,0)$	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	$\varphi(\Gamma_0+1,0,0)$	$\varphi(\varphi(\varphi(\Gamma_0+1,0,0)+1,0,0)+1,0,0)$
$\varphi(2,0,0,0)$	$\varphi(1,1,0,0)$	$\varphi(1,\varphi(1,1,0,0)+1,0,0)$	$\varphi(1,\varphi(1,\varphi(1,\varphi(1,1,0,0)+1,0,0)+1,0,0)+1,0,0)$
$\varphi(2,0,2,1)$	$\varphi(2,0,2,0)+1$	$\varphi(2,0,1,\varphi(2,0,2,0)+1)$	$\varphi(2,0,1,\varphi(2,0,1,\varphi(2,0,1,\varphi(2,0,2,0)+1)+1)+1)$
$\varphi(\omega,0,0,0)$	$\varphi(1,0,0,0)$	$\varphi(2,0,0,0)$	$\varphi(4,0,0,0)$
$\varphi(2,\varphi(2,0),\epsilon_0)$	$\varphi(2,\varphi(2,0),\omega)$	$\varphi(2,\varphi(2,0),\omega^\omega)$	$\varphi(2,\varphi(2,0),\omega^{\omega^{\omega^\omega}})$
$\varphi(3,\varphi(2,0),\epsilon_0)$	$\varphi(3,\varphi(2,0),\omega)$	$\varphi(3,\varphi(2,0),\omega^\omega)$	$\varphi(3,\varphi(2,0),\omega^{\omega^{\omega^\omega}})$
$\varphi(3,\varphi(2,0,0,0),\omega)$	$\varphi(3,\varphi(2,0,0,0),1)$	$\varphi(3,\varphi(2,0,0,0),2)$	$\varphi(3,\varphi(2,0,0,0),4)$
$\varphi(3,\varphi(2,0,0,0),\epsilon_0)$	$\varphi(3,\varphi(2,0,0,0),\omega)$	$\varphi(3,\varphi(2,0,0,0),\omega^\omega)$	$\varphi(3,\varphi(2,0,0,0),\omega^{\omega^{\omega^\omega}})$
$\varphi(3,\varphi(2,0,0,0),\epsilon_1)$	$\varphi(3,\varphi(2,0,0,0),\epsilon_0+1)$	$\varphi(3,\varphi(2,0,0,0),\omega^{\epsilon_0+1})$	$\varphi(3,\varphi(2,0,0,0),\omega^{\omega^{\omega^{\epsilon_0+1}}})$
$\varphi(\varphi(2,0),\Gamma_0,\epsilon_0)$	$\varphi(\varphi(2,0),\Gamma_0,\omega)$	$\varphi(\varphi(2,0),\Gamma_0,\omega^\omega)$	$\varphi(\varphi(2,0),\Gamma_0,\omega^{\omega^{\omega^\omega}})$

Table 12: `FiniteFuncOrdinal::limitElement` examples.

7.3 `FiniteFuncOrdinal fixedPoint` member function

`FiniteFuncOrdinal::fixedPoint` is used by `finiteFunctional` to create an instance of `FiniteFuncOrdinal` in a normal form (Equation 6) that satisfies the condition that it is the simplest expression for the ordinal represented. The routine has an index and an array of pointers to ordinal notations as input. The function determines if the parameter at the specified index is a fixed point for a `FiniteFuncOrdinal` created with the specified 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). 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 5.3.1 defined the `codeLevel` assigned to each ordinal notation. From this a `psuedoCodeLevel` is obtained by calling a member function with that name. `psuedoCodeLevel` returns `cantorCodeLevel` unless the ordinal notation normal form has a single term with factor of 1. In that case it returns the `codeLevel` of that term. 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 this `psuedoCodeLevel` $>$ `finiteFuncCodeLevel`, `true` is returned. The most significant parameter cannot be a fixed point unless it has `psuedoCodeLevel` $>$ `finiteFuncCodeLevel`. Thus, if the index selects the most significant parameter and the previous test was not passed, `false` is returned. Finally a `FiniteFuncOrdinal` 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 `FiniteFuncOrdinal` operators

`FiniteFuncOrdinal` are extensions of the `Ordinal` operators defined in Section 5.3. No new code is required for addition.

7.4.1 `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 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[25] 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`.

1. If `op1` is finite return a copy of `op2` with its factor multiplied by `op1` and return that vale.

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 a `CantorNormalElement` with exponent equal to `newExp` and `factor` equal to the `factor` or `op2`.

Some examples are shown in Table 13.

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

α	β	$\alpha \times \beta$
$\epsilon_0 + 2$	$\epsilon_0 + 2$	$\omega^{(\epsilon_0 \times 2)} + (\epsilon_0 \times 2) + 2$
$\epsilon_0 + 2$	$\epsilon_1 + \epsilon_0 + 2$	$\epsilon_1 + \omega^{(\epsilon_0 \times 2)} + (\epsilon_0 \times 2) + 2$
$\epsilon_0 + 2$	$\omega^{\omega^{(\epsilon_0 \times 2)}}$	$\omega^{\omega^{(\epsilon_0 \times 2)}}$
$\epsilon_0 + 2$	$\varphi(\omega^\omega, \omega + 3)$	$\varphi(\omega^\omega, \omega + 3)$
$\epsilon_0 + 2$	$\varphi(3, \epsilon_0, \varphi(\omega^\omega, \omega + 3))$	$\varphi(3, \epsilon_0, \varphi(\omega^\omega, \omega + 3))$
$\epsilon_1 + \epsilon_0 + 2$	$\epsilon_0 + 2$	$\omega^{\epsilon_1 + \epsilon_0} + (\epsilon_1 \times 2) + \epsilon_0 + 2$
$\epsilon_1 + \epsilon_0 + 2$	$\epsilon_1 + \epsilon_0 + 2$	$\omega^{(\epsilon_1 \times 2)} + \omega^{\epsilon_1 + \epsilon_0} + (\epsilon_1 \times 2) + \epsilon_0 + 2$
$\epsilon_1 + \epsilon_0 + 2$	$\omega^{\omega^{(\epsilon_0 \times 2)}}$	$\omega^{\epsilon_1 + \omega^{(\epsilon_0 \times 2)}}$
$\epsilon_1 + \epsilon_0 + 2$	$\varphi(\omega^\omega, \omega + 3)$	$\varphi(\omega^\omega, \omega + 3)$
$\epsilon_1 + \epsilon_0 + 2$	$\varphi(3, \epsilon_0, \varphi(\omega^\omega, \omega + 3))$	$\varphi(3, \epsilon_0, \varphi(\omega^\omega, \omega + 3))$
$\omega^{\omega^{(\epsilon_0 \times 2)}}$	$\epsilon_0 + 2$	$\omega^{\omega^{(\epsilon_0 \times 2) + \epsilon_0}} + \omega^{\omega^{(\epsilon_0 \times 2)}} \times 2$
$\omega^{\omega^{(\epsilon_0 \times 2)}}$	$\epsilon_1 + \epsilon_0 + 2$	$\epsilon_1 + \omega^{\omega^{(\epsilon_0 \times 2) + \epsilon_0}} + \omega^{\omega^{(\epsilon_0 \times 2)}} \times 2$
$\omega^{\omega^{(\epsilon_0 \times 2)}}$	$\omega^{\omega^{(\epsilon_0 \times 2)}}$	$\omega^{\omega^{(\epsilon_0 \times 2)} \times 2}$
$\omega^{\omega^{(\epsilon_0 \times 2)}}$	$\varphi(\omega^\omega, \omega + 3)$	$\varphi(\omega^\omega, \omega + 3)$
$\omega^{\omega^{(\epsilon_0 \times 2)}}$	$\varphi(3, \epsilon_0, \varphi(\omega^\omega, \omega + 3))$	$\varphi(3, \epsilon_0, \varphi(\omega^\omega, \omega + 3))$
$\varphi(\omega^\omega, \omega + 3)$	$\epsilon_0 + 2$	$\omega^{\varphi(\omega^\omega, \omega+3) + \epsilon_0} + (\varphi(\omega^\omega, \omega + 3) \times 2)$
$\varphi(\omega^\omega, \omega + 3)$	$\epsilon_1 + \epsilon_0 + 2$	$\omega^{\varphi(\omega^\omega, \omega+3) + \epsilon_1} + \omega^{\varphi(\omega^\omega, \omega+3) + \epsilon_0} + (\varphi(\omega^\omega, \omega + 3) \times 2)$
$\varphi(\omega^\omega, \omega + 3)$	$\omega^{\omega^{(\epsilon_0 \times 2)}}$	$\omega^{\varphi(\omega^\omega, \omega+3) + \omega^{(\epsilon_0 \times 2)}}$
$\varphi(\omega^\omega, \omega + 3)$	$\varphi(\omega^\omega, \omega + 3)$	$\omega^{(\varphi(\omega^\omega, \omega+3) \times 2)}$
$\varphi(\omega^\omega, \omega + 3)$	$\varphi(3, \epsilon_0, \varphi(\omega^\omega, \omega + 3))$	$\varphi(3, \epsilon_0, \varphi(\omega^\omega, \omega + 3))$
$\varphi(3, \epsilon_0, \varphi(\omega^\omega, \omega + 3))$	$\epsilon_0 + 2$	$\omega^{\varphi(3, \epsilon_0, \varphi(\omega^\omega, \omega+3)) + \epsilon_0} + (\varphi(3, \epsilon_0, \varphi(\omega^\omega, \omega + 3)) \times 2)$
$\varphi(3, \epsilon_0, \varphi(\omega^\omega, \omega + 3))$	$\epsilon_1 + \epsilon_0 + 2$	$\omega^{\varphi(3, \epsilon_0, \varphi(\omega^\omega, \omega+3)) + \epsilon_1} + \omega^{\varphi(3, \epsilon_0, \varphi(\omega^\omega, \omega+3)) + \epsilon_0} + (\varphi(3, \epsilon_0, \varphi(\omega^\omega, \omega + 3)) \times 2)$
$\varphi(3, \epsilon_0, \varphi(\omega^\omega, \omega + 3))$	$\omega^{\omega^{(\epsilon_0 \times 2)}}$	$\omega^{\varphi(3, \epsilon_0, \varphi(\omega^\omega, \omega+3)) + \omega^{(\epsilon_0 \times 2)}}$
$\varphi(3, \epsilon_0, \varphi(\omega^\omega, \omega + 3))$	$\varphi(\omega^\omega, \omega + 3)$	$\omega^{\varphi(3, \epsilon_0, \varphi(\omega^\omega, \omega+3)) + \varphi(\omega^\omega, \omega+3)}$
$\varphi(3, \epsilon_0, \varphi(\omega^\omega, \omega + 3))$	$\varphi(3, \epsilon_0, \varphi(\omega^\omega, \omega + 3))$	$\omega^{(\varphi(3, \epsilon_0, \varphi(\omega^\omega, \omega+3)) \times 2)}$

Table 13: `FiniteFuncOrdinal` multiply examples

7.4.2 `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. $a = \omega^a$. Define $f(a)$ (where a is a single term in the normal form of an ordinal) as equal to $a/a.\texttt{factor}$. $f(a)$ Thus $f(a)$ sets the `factor` part of the term to 1. 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 $\omega^{f(a)b}$ or equivalently $\omega^{\omega^{f(a)+b}}$ which is the normal form result. If the base, a, of the exponential is at `cantorCodeLevel` then $a = \omega^\alpha$ and the result is $\omega^{\alpha b}$.

The above describes the bulk of the algorithm in which the base of the exponentiation is `base` and the exponent is `expon`. At the start of the algorithm if `expon.maxNotationLevel` > `base.maxNotationLevel` the result is `expon`. As with multiplication if the exponent of the result, `newExp`, has a single term in its normal form and that term is at `finiteFuncCodeLevel` with a `factor` of 1 then `newExp` is returned instead of $\omega^{\texttt{newExp}}$. Some examples are shown in Table 14.

α	β	α^β
ϵ_0+2	ϵ_0+2	$\omega^{\omega^{(\epsilon_0\times 2)}+(\epsilon_0\times 2)}+\omega^{\omega^{(\epsilon_0\times 2)}+\epsilon_0}\times 2+\omega^{\omega^{(\epsilon_0\times 2)}}\times 2$
ϵ_0+2	$\epsilon_1+\epsilon_0$	$\omega^{\epsilon_1+\omega^{(\epsilon_0\times 2)}}$
ϵ_0+2	$\omega^{\omega^{(\epsilon_0\times 2)}}+3$	$\omega^{\omega^{\omega^{(\epsilon_0\times 2)}}+(\epsilon_0\times 3)}+\omega^{\omega^{\omega^{(\epsilon_0\times 2)}}+(\epsilon_0\times 2)}\times 2+\omega^{\omega^{\omega^{(\epsilon_0\times 2)}}+\epsilon_0}\times 2+\omega^{\omega^{\omega^{(\epsilon_0\times 2)}}}\times 2$
ϵ_0+2	$\varphi(\omega^\omega,\omega+3)$	$\varphi(\omega^\omega,\omega+3)$
ϵ_0+2	$\varphi(3,\epsilon_0,\varphi(\omega^\omega,\omega+3))$	$\varphi(3,\epsilon_0,\varphi(\omega^\omega,\omega+3))$
$\epsilon_1+\epsilon_0$	ϵ_0+2	$\omega^{\omega^{\epsilon_1+\epsilon_0}+(\epsilon_1\times 2)}+\omega^{\omega^{\epsilon_1+\epsilon_0}+\epsilon_1+\epsilon_0}$
$\epsilon_1+\epsilon_0$	$\epsilon_1+\epsilon_0$	$\omega^{\omega^{(\epsilon_1\times 2)}+\omega^{\epsilon_1+\epsilon_0}}$
$\epsilon_1+\epsilon_0$	$\omega^{\omega^{(\epsilon_0\times 2)}}+3$	$\omega^{\omega^{\epsilon_1+\omega^{(\epsilon_0\times 2)}}+(\epsilon_1\times 3)}+\omega^{\omega^{\epsilon_1+\omega^{(\epsilon_0\times 2)}}+(\epsilon_1\times 2)+\epsilon_0}$
$\epsilon_1+\epsilon_0$	$\varphi(\omega^\omega,\omega+3)$	$\varphi(\omega^\omega,\omega+3)$
$\epsilon_1+\epsilon_0$	$\varphi(3,\epsilon_0,\varphi(\omega^\omega,\omega+3))$	$\varphi(3,\epsilon_0,\varphi(\omega^\omega,\omega+3))$
$\omega^{\omega^{(\epsilon_0\times 2)}}+3$	ϵ_0+2	$\omega^{\omega^{(\epsilon_0\times 3)}+\omega^{(\epsilon_0\times 2)}\times 2}+\omega^{\omega^{(\epsilon_0\times 3)}+\omega^{(\epsilon_0\times 2)}}\times 3+\omega^{\omega^{(\epsilon_0\times 3)}}\times 3$
$\omega^{\omega^{(\epsilon_0\times 2)}}+3$	$\epsilon_1+\epsilon_0$	$\omega^{\epsilon_1+\omega^{(\epsilon_0\times 3)}}$
$\omega^{\omega^{(\epsilon_0\times 2)}}+3$	$\omega^{\omega^{(\epsilon_0\times 2)}}+3$	$\omega^{\omega^{\omega^{(\epsilon_0\times 2)}}+\omega^{(\epsilon_0\times 2)}\times 3}+\omega^{\omega^{\omega^{(\epsilon_0\times 2)}}+\omega^{(\epsilon_0\times 2)}\times 2}\times 3+\omega^{\omega^{\omega^{(\epsilon_0\times 2)}}+\omega^{(\epsilon_0\times 2)}}\times 3+\omega^{\omega^{\omega^{(\epsilon_0\times 2)}}}\times 3$
$\omega^{\omega^{(\epsilon_0\times 2)}}+3$	$\varphi(\omega^\omega,\omega+3)$	$\varphi(\omega^\omega,\omega+3)$
$\omega^{\omega^{(\epsilon_0\times 2)}}+3$	$\varphi(3,\epsilon_0,\varphi(\omega^\omega,\omega+3))$	$\varphi(3,\epsilon_0,\varphi(\omega^\omega,\omega+3))$
$\varphi(\omega^\omega,\omega+3)$	ϵ_0+2	$\omega^{\omega^{\varphi(\omega^\omega,\omega+3)+\epsilon_0}+(\varphi(\omega^\omega,\omega+3)\times 2)}$
$\varphi(\omega^\omega,\omega+3)$	$\epsilon_1+\epsilon_0$	$\omega^{\omega^{\varphi(\omega^\omega,\omega+3)+\epsilon_1}+\omega^{\varphi(\omega^\omega,\omega+3)+\epsilon_0}}$
$\varphi(\omega^\omega,\omega+3)$	$\omega^{\omega^{(\epsilon_0\times 2)}}+3$	$\omega^{\omega^{\varphi(\omega^\omega,\omega+3)+\omega^{(\epsilon_0\times 2)}}+(\varphi(\omega^\omega,\omega+3)\times 3)}$
$\varphi(\omega^\omega,\omega+3)$	$\varphi(\omega^\omega,\omega+3)$	$\omega^{\omega^{(\varphi(\omega^\omega,\omega+3)\times 2)}}$
$\varphi(\omega^\omega,\omega+3)$	$\varphi(3,\epsilon_0,\varphi(\omega^\omega,\omega+3))$	$\varphi(3,\epsilon_0,\varphi(\omega^\omega,\omega+3))$
$\varphi(3,\epsilon_0,\varphi(\omega^\omega,\omega+3))$	ϵ_0+2	$\omega^{\omega^{\varphi(3,\epsilon_0,\varphi(\omega^\omega,\omega+3))+\epsilon_0}+(\varphi(3,\epsilon_0,\varphi(\omega^\omega,\omega+3))\times 2)}$
$\varphi(3,\epsilon_0,\varphi(\omega^\omega,\omega+3))$	$\epsilon_1+\epsilon_0$	$\omega^{\omega^{\varphi(3,\epsilon_0,\varphi(\omega^\omega,\omega+3))+\epsilon_1}+\omega^{\varphi(3,\epsilon_0,\varphi(\omega^\omega,\omega+3))+\epsilon_0}}$
$\varphi(3,\epsilon_0,\varphi(\omega^\omega,\omega+3))$	$\omega^{\omega^{(\epsilon_0\times 2)}}+3$	$\omega^{\omega^{\varphi(3,\epsilon_0,\varphi(\omega^\omega,\omega+3))+\omega^{(\epsilon_0\times 2)}}+(\varphi(3,\epsilon_0,\varphi(\omega^\omega,\omega+3))\times 3)}$
$\varphi(3,\epsilon_0,\varphi(\omega^\omega,\omega+3))$	$\varphi(\omega^\omega,\omega+3)$	$\omega^{\omega^{\varphi(3,\epsilon_0,\varphi(\omega^\omega,\omega+3))+\varphi(\omega^\omega,\omega+3)}}$
$\varphi(3,\epsilon_0,\varphi(\omega^\omega,\omega+3))$	$\varphi(3,\epsilon_0,\varphi(\omega^\omega,\omega+3))$	$\omega^{\omega^{(\varphi(3,\epsilon_0,\varphi(\omega^\omega,\omega+3))\times 2)}}$

Table 14: `FiniteFuncOrdinal` exponential examples

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
'cp' stands for function `createParameters`	

Table 15: `IterFuncOrdinal` C++ code examples

8 `IterFuncOrdinal class`

C++ `class IterFuncOrdinal` is derived from `class FiniteFuncOrdinal` which in turn is derived from `class Ordinal`. It implements the iterative function hierarchy described in Section 6.5. 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`.

The `IterFuncOrdinal class` should not be used directly to create ordinal notations. Instead use function `iterativeFunctional`. 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 `Ordinal`s which are the $\beta_{i,j}$ in Equation 7. This second parameter is optional and can be created with function `createParameters` described in Section 7. Some examples are shown in Table 15.

`iterativeFunctional` declares an `Ordinal` or `FiniteFuncOrdinal` instead of an `IterFuncOrdinal` if appropriate. The first three lines of Table 15 are examples. It also reduces fixed points to their simplest possible expression. The last line of the table is an example.

8.1 `IterFuncOrdinal compare` member function

`IterFuncOrdinal::compare` supports the extended normal form in Equation 7. The `compare virtual` functions overridden by `class`es `IterFuncOrdinal` and `IterFuncNormalElement` are those that compare terms of a normal form, `CantorNormalElement`s, and a single term against a full ordinal notation, `class`

OrdinalImpl(see note 22). The latter compare function is trivial except for its dependence on the former.

IterFuncNormalElement::compare with a single CantorNormalElement argument is the main function that implements compare. It overrides FiniteFuncNormalElement::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 term.

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[26] if IterFuncNormalElement then -trm.compare(*this) is returned. This invokes the subclass member function associated with the subclass and codeLevel of trm.

This compare is similar to that for class FiniteFuncOrdinal described in Section 7.1. 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 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.

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 factors). 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 result is nonzero that result 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 this routine returns that the parameters of the two terms are identical the difference in the factors of the two terms is returned.

[26] iterFuncCodeLevel is the codeLevel (see Section 5.3.2) of an IterFuncNormalElement.

8.2 `IterFuncOrdinal limitElement` member function

`IterFuncOrdinal::limitElement` overrides `Ordinal::limitElement` described in Section 5.2 and `FiniteFuncOrdinal::limitElement` described in Section 7.2 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 `IterFuncOrdinal class` instance `limitElement` is called from. This will be referred to as the input `Ordinal` to `limitElement`.

As with `Ordinal::limitElement`, all but the last term of the normal form of the result are copied 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 `Ordinal`.

Tables 2, 9 and 10 fully define `IterFuncOrdinal::limitElement`.[27] 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`[28] of tables 9 and 10 also always call this `virtual` function to create a result.

Some examples are shown in Table 16.

[27]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 letter as a parameter. This letter is an exit code that matches the X column in the tables.

[28]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
$\varphi_\omega(1)$	$\varphi_\omega+1$	$\varphi_2(\varphi_\omega+1,0)$	$\varphi_3(\varphi_\omega+1,0,0)$
$\varphi_1(2)$	$\omega^{\varphi_1(1)+1}$	$\varphi(\varphi_1(1)+1,0)$	$\varphi(\varphi_1(1)+1,0,0)$
$\varphi_1(\epsilon_0)$	$\varphi_1(\omega)$	$\varphi_1(\omega^\omega)$	$\varphi_1(\omega^{\omega^\omega})$
$\varphi_1(1,0,0)$	$\varphi_1(1,0)$	$\varphi_1(\varphi_1(1,0)+1,0)$	$\varphi_1(\varphi_1(\varphi_1(1,0)+1,0)+1,0)$
$\varphi_3(1,0,0)$	$\varphi_3(1,0)$	$\varphi_3(\varphi_3(1,0)+1,0)$	$\varphi_3(\varphi_3(\varphi_3(1,0)+1,0)+1,0)$
$\varphi_1(\omega,1)$	$\varphi_1(1,\varphi_1(\omega,0)+1)$	$\varphi_1(2,\varphi_1(\omega,0)+1)$	$\varphi_1(3,\varphi_1(\omega,0)+1)$
$\varphi_1(\omega,1,0)$	$\varphi_1(\omega,0,1)$	$\varphi_1(\omega,0,\varphi_1(\omega,0,1)+1)$	$\varphi_1(\omega,0,\varphi_1(\omega,0,\varphi_1(\omega,0,1)+1)+1)$
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}	φ_{ϵ_0+1}	$\varphi_{\varphi(\epsilon_0+1,0)+1}$	$\varphi_{\varphi(\varphi(\epsilon_0+1,0)+1,0)+1}$
$\varphi_{\Gamma_0}(1)$	$\varphi_{\Gamma_0}+1$	$\varphi_{\varphi(\epsilon_0+1,0)+1}(\varphi_{\Gamma_0}+1,0)$	$\varphi_{\varphi(\varphi(\epsilon_0+1,0)+1,0)+1}(\varphi_{\Gamma_0}+1,0,0)$
$\varphi_{\Gamma_0}(1,0)$	$\varphi_{\Gamma_0}(1)$	$\varphi_{\Gamma_0}(\varphi_{\Gamma_0}(1)+1)$	$\varphi_{\Gamma_0}(\varphi_{\Gamma_0}(\varphi_{\Gamma_0}(1)+1)+1)$
$\varphi_{\Gamma_0}(\epsilon_0)$	$\varphi_{\Gamma_0}(\omega)$	$\varphi_{\Gamma_0}(\omega^\omega)$	$\varphi_{\Gamma_0}(\omega^{\omega^\omega})$
$\varphi_{\Gamma_0}(\epsilon_0,0)$	$\varphi_{\Gamma_0}(\omega,0)$	$\varphi_{\Gamma_0}(\omega^\omega,0)$	$\varphi_{\Gamma_0}(\omega^{\omega^\omega},0)$
$\varphi_{\Gamma_0}(\epsilon_0,1)$	$\varphi_{\Gamma_0}(\omega,\varphi_{\Gamma_0}(\epsilon_0,0)+1)$	$\varphi_{\Gamma_0}(\omega^\omega,\varphi_{\Gamma_0}(\epsilon_0,0)+1)$	$\varphi_{\Gamma_0}(\omega^{\omega^\omega},\varphi_{\Gamma_0}(\epsilon_0,0)+1)$
$\varphi_{\Gamma_0}(\epsilon_0,1,1)$	$\varphi_{\Gamma_0}(\epsilon_0,1,0)+1$	$\varphi_{\Gamma_0}(\epsilon_0,0,\varphi_{\Gamma_0}(\epsilon_0,1,0)+1)$	$\varphi_{\Gamma_0}(\epsilon_0,0,\varphi_{\Gamma_0}(\epsilon_0,0,\varphi_{\Gamma_0}(\epsilon_0,1,0)+1)+1)$
$\varphi_{\Gamma_0}(\epsilon_0,1,1,2)$	$\varphi_{\Gamma_0}(\epsilon_0,1,1,1)+1$	$\varphi_{\Gamma_0}(\epsilon_0,1,0,\varphi_{\Gamma_0}(\epsilon_0,1,1,1)+1)$	$\varphi_{\Gamma_0}(\epsilon_0,1,0,\varphi_{\Gamma_0}(\epsilon_0,1,0,\varphi_{\Gamma_0}(\epsilon_0,1,1,1)+1)+1)$

Table 16: `IterFuncOrdinal::limitElement` examples.

8.3 `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.

- The function level or γ from a single term in Equation 7.
- 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 `iterFirstNz` defined as -1 in an `enum`.
- The function parameters as an array of pointers to `Ordinal`s. 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 `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 7.3 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`.

8.4 `IterFuncOrdinal` operators

No new code is required for multiplication and exponentiation. The routines for `FiniteFuncOrdinal` and `Ordinal` and the associated `class`es for normal form terms `CantorNormalElement` and `FiniteFuncNormalElement` do not need to be overrides except for some 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 17. Some exponential examples shown in Table 18.

α	β	$\alpha \times \beta$
φ_1	φ_1	$\omega^{(\varphi_1\times 2)}$
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	$\varphi_1(\omega,1,0)$	$\varphi_1(\omega,1,0)$
φ_1	φ_{ϵ_0}	φ_{ϵ_0}
φ_3	φ_1	$\omega^{\varphi_3+\varphi_1}$
φ_3	φ_3	$\omega^{(\varphi_3\times 2)}$
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	$\varphi_1(\omega,1,0)$	$\omega^{\varphi_3+\varphi_1(\omega,1,0)}$
φ_3	φ_{ϵ_0}	φ_{ϵ_0}
$\varphi_3(1,0,0)$	φ_1	$\omega^{\varphi_3(1,0,0)+\varphi_1}$
$\varphi_3(1,0,0)$	φ_3	$\omega^{\varphi_3(1,0,0)+\varphi_3}$
$\varphi_3(1,0,0)$	$\varphi_3(1,0,0)$	$\omega^{(\varphi_3(1,0,0)\times 2)}$
$\varphi_3(1,0,0)$	$\varphi_1(\omega,1)$	$\omega^{\varphi_3(1,0,0)+\varphi_1(\omega,1)}$
$\varphi_3(1,0,0)$	$\varphi_1(\omega,1,0)$	$\omega^{\varphi_3(1,0,0)+\varphi_1(\omega,1,0)}$
$\varphi_3(1,0,0)$	φ_{ϵ_0}	φ_{ϵ_0}
$\varphi_1(\omega,1)$	φ_1	$\omega^{\varphi_1(\omega,1)+\varphi_1}$
$\varphi_1(\omega,1)$	φ_3	φ_3
$\varphi_1(\omega,1)$	$\varphi_3(1,0,0)$	$\varphi_3(1,0,0)$
$\varphi_1(\omega,1)$	$\varphi_1(\omega,1)$	$\omega^{(\varphi_1(\omega,1)\times 2)}$
$\varphi_1(\omega,1)$	$\varphi_1(\omega,1,0)$	$\varphi_1(\omega,1,0)$
$\varphi_1(\omega,1)$	φ_{ϵ_0}	φ_{ϵ_0}
$\varphi_1(\omega,1,0)$	φ_1	$\omega^{\varphi_1(\omega,1,0)+\varphi_1}$
$\varphi_1(\omega,1,0)$	φ_3	φ_3
$\varphi_1(\omega,1,0)$	$\varphi_3(1,0,0)$	$\varphi_3(1,0,0)$
$\varphi_1(\omega,1,0)$	$\varphi_1(\omega,1)$	$\omega^{\varphi_1(\omega,1,0)+\varphi_1(\omega,1)}$
$\varphi_1(\omega,1,0)$	$\varphi_1(\omega,1,0)$	$\omega^{(\varphi_1(\omega,1,0)\times 2)}$
$\varphi_1(\omega,1,0)$	φ_{ϵ_0}	φ_{ϵ_0}
φ_{ϵ_0}	φ_1	$\omega^{\varphi_{\epsilon_0}+\varphi_1}$
φ_{ϵ_0}	φ_3	$\omega^{\varphi_{\epsilon_0}+\varphi_3}$
φ_{ϵ_0}	$\varphi_3(1,0,0)$	$\omega^{\varphi_{\epsilon_0}+\varphi_3(1,0,0)}$
φ_{ϵ_0}	$\varphi_1(\omega,1)$	$\omega^{\varphi_{\epsilon_0}+\varphi_1(\omega,1)}$
φ_{ϵ_0}	$\varphi_1(\omega,1,0)$	$\omega^{\varphi_{\epsilon_0}+\varphi_1(\omega,1,0)}$
φ_{ϵ_0}	φ_{ϵ_0}	$\omega^{(\varphi_{\epsilon_0}\times 2)}$

Table 17: `IterFuncOrdinal` multiply examples

α	β	α^β
φ_1	φ_1	$\omega^{\omega^{(\varphi_1\times 2)}}$
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	$\varphi_1(\omega,1,0)$	$\varphi_1(\omega,1,0)$
φ_1	φ_{ϵ_0}	φ_{ϵ_0}
φ_3	φ_1	$\omega^{\omega^{\varphi_3+\varphi_1}}$
φ_3	φ_3	$\omega^{\omega^{(\varphi_3\times 2)}}$
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	$\varphi_1(\omega,1,0)$	$\omega^{\omega^{\varphi_3+\varphi_1(\omega,1,0)}}$
φ_3	φ_{ϵ_0}	φ_{ϵ_0}
$\varphi_3(1,0,0)$	φ_1	$\omega^{\omega^{\varphi_3(1,0,0)+\varphi_1}}$
$\varphi_3(1,0,0)$	φ_3	$\omega^{\omega^{\varphi_3(1,0,0)+\varphi_3}}$
$\varphi_3(1,0,0)$	$\varphi_3(1,0,0)$	$\omega^{\omega^{(\varphi_3(1,0,0)\times 2)}}$
$\varphi_3(1,0,0)$	$\varphi_1(\omega,1)$	$\omega^{\omega^{\varphi_3(1,0,0)+\varphi_1(\omega,1)}}$
$\varphi_3(1,0,0)$	$\varphi_1(\omega,1,0)$	$\omega^{\omega^{\varphi_3(1,0,0)+\varphi_1(\omega,1,0)}}$
$\varphi_3(1,0,0)$	φ_{ϵ_0}	φ_{ϵ_0}
$\varphi_1(\omega,1)$	φ_1	$\omega^{\omega^{\varphi_1(\omega,1)+\varphi_1}}$
$\varphi_1(\omega,1)$	φ_3	φ_3
$\varphi_1(\omega,1)$	$\varphi_3(1,0,0)$	$\varphi_3(1,0,0)$
$\varphi_1(\omega,1)$	$\varphi_1(\omega,1)$	$\omega^{\omega^{(\varphi_1(\omega,1)\times 2)}}$
$\varphi_1(\omega,1)$	$\varphi_1(\omega,1,0)$	$\varphi_1(\omega,1,0)$
$\varphi_1(\omega,1)$	φ_{ϵ_0}	φ_{ϵ_0}
$\varphi_1(\omega,1,0)$	φ_1	$\omega^{\omega^{\varphi_1(\omega,1,0)+\varphi_1}}$
$\varphi_1(\omega,1,0)$	φ_3	φ_3
$\varphi_1(\omega,1,0)$	$\varphi_3(1,0,0)$	$\varphi_3(1,0,0)$
$\varphi_1(\omega,1,0)$	$\varphi_1(\omega,1)$	$\omega^{\omega^{\varphi_1(\omega,1,0)+\varphi_1(\omega,1)}}$
$\varphi_1(\omega,1,0)$	$\varphi_1(\omega,1,0)$	$\omega^{\omega^{(\varphi_1(\omega,1,0)\times 2)}}$
$\varphi_1(\omega,1,0)$	φ_{ϵ_0}	φ_{ϵ_0}
φ_{ϵ_0}	φ_1	$\omega^{\omega^{\varphi_{\epsilon_0}+\varphi_1}}$
φ_{ϵ_0}	φ_3	$\omega^{\omega^{\varphi_{\epsilon_0}+\varphi_3}}$
φ_{ϵ_0}	$\varphi_3(1,0,0)$	$\omega^{\omega^{\varphi_{\epsilon_0}+\varphi_3(1,0,0)}}$
φ_{ϵ_0}	$\varphi_1(\omega,1)$	$\omega^{\omega^{\varphi_{\epsilon_0}+\varphi_1(\omega,1)}}$
φ_{ϵ_0}	$\varphi_1(\omega,1,0)$	$\omega^{\omega^{\varphi_{\epsilon_0}+\varphi_1(\omega,1,0)}}$
φ_{ϵ_0}	φ_{ϵ_0}	$\omega^{\omega^{(\varphi_{\epsilon_0}\times 2)}}$

Table 18: `IterFuncOrdinal` exponential examples

9 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[2, 1]. 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 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 in powerful ways. Thus it is consistent but, I suspect, incorrect to reason as if completed infinite totalities exist. There is a recursive process that will enumerate the names of all the reals definable in any formal mathematical system, as Lowenheim and Skolem observed in the theorem that bears their names, and 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[3].

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

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 the Veblen function with any number of parameters and one step beyond that. Following are topics you can get more information about by entering 'help topic'.

'`cmds`' – lists commands.
'`compare`' – describes comparison operators.
'`members`' – describes member functions.
'`ordinal`' – describes available ordinal notations.
'`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.2 Ordinals

Ordinals are displayed in TeX and plain text format. (Enter 'help opts' to see how to control this.) The plain text format can be read. 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 lower case 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(1,0,x) is displayed as gamma(x). In all cases the displayed version can be used as input.

The largest ordinals in this implementation 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.

A.3 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. 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.4 Commands

A.4.1 All commands

The following commands are available:
'`examples`' – shows examples
'`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
'`quit`' – exits the program

'`read`' – reads an input file (ord_calc.ord default)
'`save`' – saves assignment statements to a file (ord_calc.ord default)
'`yydebug`' – enables parser debugging (off option)
Type 'help command_name' to learn more about a specific command.

A.4.2 Commands with options

Following are the commands with options.

Command '`examples`' – shows examples.
It has one parameter with the following options.
'`arith`' – show a simple ordinal arithmetic example.
'`compare`' – show some compare examples.
'`display`' – show display options examples.
'`member`' – show member function examples.

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).

A.5 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 a sequence of 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.

The member functions are:

'`limitElt`' – 'evaluates to specified (no default) limit element'
'`listLimitElts`' – 'lists specified (default 10) limit elements'

A.6 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.7 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`' – show a simple ordinal arithmetic example
'`compare`' – show some compare examples
'`display`' – show display options examples
'`member`' – show member function examples

A.7.1 Simple 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.7.2 Comparison operators

```
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.7.3 Display options

```
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 = \epsilon_{\omega{}^{\omega{}^{\omega{}}}}
c = \varphi( 1, 0, \epsilon_{\omega{}^{\omega{}^{\omega{}}}})
ordCalc>opts both
ordCalc>list
a = (w^((w^(w))))
a = \omega{}^{\omega{}^{\omega{}}}
b = epsilon((w^((w^(w)))))
b = \epsilon_{\omega{}^{\omega{}^{\omega{}}}}
c = gamma(epsilon((w^((w^(w)))))))
c = \varphi( 1, 0, \epsilon_{\omega{}^{\omega{}^{\omega{}}}})
```

A.7.4 Member functions

```
ordCalc>a=psi(1,0,0,0,0)
Assigning psi(1, 0, 0, 0, 0) to 'a'.
ordCalc>a.listLimitElts(3)


First 3 limitElements for psi(1, 0, 0, 0, 0)
le(0) = 0
le(1) = psi(1, 0, 0, 0)
```

```
le(2) = psi(psi(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’.
```

References

[1] Paul Budnik. *What is and what will be: Integrating spirituality and science.* Mountain Math Software, Los Gatos, CA, 2006. 39

[2] Paul Budnik. What is Mathematics About? *Philosophy of Mathematics Education*, 22, 2007. 39

[3] Solomon Feferman. Does mathematics need new axioms? *American Mathematical Monthly*, 106:99–111, 1999. 40

[4] 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. 6, 17

[5] Larry W. Miller. Normal functions and constructive ordinal notations. *The Journal of Symbolic Logic*, 41(2):439–459, 1976. 6, 17

[6] Oswald Veblen. Continuous increasing functions of finite and transfinite ordinals. *Transactions of the American Mathematical Society*, 9(3):280–292, 1908. 6, 17

Formatted: September 15, 2009