A Computational Approach to the Ordinal Numbers
Documents ordCalc_0.3

Paul Budnik
Mountain Math Software
paul@mtnmath.com

Copyright © 2009 - 2011 Mountain Math Software
All Rights Reserved
Licensed for use and distribution under
GNU General Public License Version 2.

Source code and documentation can be downloaded at
www.mtnmath.com/ord
and sourceforge.net/projects/ord.

Contents

List of Tables 4

1 Introduction 6
   1.1 Intended audience ........................................ 6
   1.2 The Ordinals ........................................... 7

2 Ordinal notations 8
   2.1 Ordinal functions and fixed points ......................... 9
   2.2 Beyond the recursive ordinals .......................... 10
   2.3 Uncountable ordinals ................................ 10

3 Program structure 11
   3.1 virtual functions and subclasses ........................ 12
   3.2 Ordinal normal forms .................................. 12
   3.3 Memory management .................................... 13

4 Ordinal base class 13
   4.1 normalForm and texNormalForm member functions ........ 14
   4.2 compare member function ................................ 14
   4.3 limitElement member function ........................... 14
   4.4 Operators ............................................. 15
       4.4.1 Addition ........................................ 16
11 NestedEmbedOrdinal class

11.1 compare member function ........................................ 88
11.2 class NestedEmbeddings .............................................. 91
  11.2.1 compare member function ..................................... 91
  11.2.2 nextLeast member function .................................. 91
11.3 limitElement member function ................................... 91
11.4 isValidLimitOrdParam and maxLimitType member functions .... 102
11.5 limitInfo, limitType and embedType member functions ....... 102
11.6 limitOrd member function ......................................... 102
11.7 fixedPoint member function ....................................... 102

12 Philosophical Issues .................................................. 102

A Formalizing objective mathematics .................................. 107
A.1 Introduction .................................................................. 107
  A.1.1 The mathematics of recursive processes ...................... 107
  A.1.2 The uncountable .................................................... 107
  A.1.3 Expanding the foundations of mathematics .................. 108
A.2 Background .................................................................... 108
  A.2.1 The ordinal hierarchy .............................................. 109
A.3 The true power set ...................................................... 109
A.4 Mathematical Objects ................................................. 110
  A.4.1 Properties of properties .......................................... 111
  A.4.2 Gödel and mathematical creativity ............................ 111
  A.4.3 Cantor’s incompleteness proof .................................. 111
A.5 Axioms of ZFC ............................................................ 111
  A.5.1 Axiom of extensionality .......................................... 112
  A.5.2 Axiom of the empty set .......................................... 112
  A.5.3 Axiom of unordered pairs ....................................... 112
  A.5.4 Axiom of union ..................................................... 112
  A.5.5 Axiom of infinity ................................................... 113
  A.5.6 Axiom scheme of replacement .................................. 113
  A.5.7 Power set axiom .................................................... 114
  A.5.8 Axiom of Choice .................................................... 114
A.6 The axioms of ZFC summary ......................................... 114
A.7 The Objective Parts of ZF ............................................. 115
A.8 Formalization of the Objective Parts of ZF ....................... 116
  A.8.1 Axioms unchanged from ZF ..................................... 116
  A.8.2 Objective axiom of replacement ................................ 116
A.9 An Objective Interpretation of ZFC ................................ 117
A.10 A Creative Philosophy of Mathematics ............................ 117
13 FiniteFuncOrdinal C++ code examples ................................... 27
14 FiniteFuncNormalElement::limitElementCom cases ...................... 29
15 FiniteFuncNormalElement::limitElementCom exit codes .................. 30
16 FiniteFuncOrdinal multiply examples ..................................... 33
17 FiniteFuncOrdinal exponential examples ................................ 35
18 FiniteFuncNormalElement::limitOrdCom ................................ 36
19 iterativeFunctional C++ code examples .................................. 37
20 IterFuncNormalElement::limitElementCom cases ....................... 39
21 IterFuncNormalElement::limitElementCom exit codes ................. 40
22 IterFuncOrdinal multiply examples ....................................... 42
23 IterFuncOrdinal exponential examples .................................... 43
24 limitType and maxLimitType examples .................................... 47
25 Equations for $[\delta], [\eta]$ and $[[\eta]]$ ................................. 48
26 $[[\delta]], [\eta]$ and $[[\eta]]$ parameter examples in increasing order ... 48
27 $\Psi$ collapsing function with bounds .................................... 51
28 $\Psi$ collapsing function at critical limits ................................ 52
29 Structure of $[[1]]\omega_1$ ..................................................... 55
30 admisLevelFunctional C++ code examples ................................. 58
31 Symbols used in compare tables 32 and 33 .............................. 59
32 AdmisNormalElement::compare summary .................................. 60
33 AdmisNormalElement::compareCom summary ............................. 60
34 LimitTypeInfo descriptions through admisCodeLevel ................... 62
35 Symbols used in limitElement Tables 36 to 39 and limitOrd Table 48 .. 63
36 AdmisNormalElement::limitElement cases ................................ 64
37 AdmisNormalElement::limitElementCom cases ........................... 64
38 AdmisNormalElement::drillDownLimitElement cases .................... 65
39 AdmisNormalElement::drillDownLimitElementCom cases .............. 65
40 AdmisNormalElement::limitElement exit codes ........................... 66
41 AdmisNormalElement::limitElementCom exit codes ..................... 67
42 AdmisNormalElement::drillDownLimitElement exit codes .............. 67
43 AdmisNormalElement::drillDownLimitElementCom exit codes .......... 67
44 limitElement and LimitTypeInfo examples through admisCodeLevel .... 69
45 AdmisLevOrdinal::limitElement examples part 1 .......................... 70
46 AdmisLevOrdinal::limitElement examples part 2 ........................ 71
47 AdmisLevOrdinal::limitElement examples part 3 ........................ 72
48 AdmisNormalElement::limitOrdCom cases ................................ 75
49 AdmisNormalElement::limitOrdCom exit codes ........................... 76
50 limitOrd and LimitTypeInfo examples through admisCodeLevel ....... 77
51 AdmisLevOrdinal::limitOrd examples ..................................... 78
52 LimitTypeInfo descriptions through nestedEmbedCodeLevel .......... 86
53 limitElement and LimitTypeInfo examples through nestedEmbedCodeLevel 87
54 NestedEmbedOrdinal interpreter code examples .......................... 90
55 Symbols used in compare Table 56 ....................................... 92
56 NestedEmbedNormalElement::compare summary .......................... 92
57 NestedEmbedNormalElement::limitElement cases ........................ 94
1 Introduction

The ordinal calculator is a tool for learning about the ordinal hierarchy and ordinal notations. It is also a research tool. Its motivating goal is ultimately to expand the foundations of mathematics by using computer technology to manage the combinatorial explosion in complexity that comes with explicitly defining the recursive ordinals implicitly defined by the axioms of Zermelo Frankel set theory [6, 3]. The underlying philosophy focuses on what formal systems tell us about physically realizable combinatorial processes [4]. Appendix A elaborates on this.

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

All the source code and documentation (including this manual) is licensed for use and distribution under the GNU General Public License, version 2. The executables link to some libraries that are covered by the more restrictive version 3 of this license.

1.1 Intended audience

This document is targeted to mathematicians with limited experience in computer programming and computer scientists with limited knowledge of the foundations of mathematics. Thus it contains substantial tutorial material, often as footnotes. The ideas in this paper have been implemented in the C++ programming language. C++ keywords and constructs 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 B, one can use most of the facilities of the program interactively with no knowledge of C++.

1.2 The Ordinals

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

The smallest ordinals are the integers. Other ordinals are defined as infinite sets\(^2\). The smallest infinite ordinal is the set of all integers. Infinite objects are not subject to computational manipulation. However the set of all integers can be represented by a computer program that lists the integers. This is an abstraction. Real programs cannot run forever error free. However the program itself is a finite object, a set of instructions, that a computer program can manipulate and transform. Ordinals at or beyond \(\omega\) 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 \(\alpha\), enumerates notations of all smaller ordinals. This latter algorithm cannot exist for ordinals that are not recursive but an incomplete variant of it can be defined for countable ordinals.

The ultimate goal of this research is to construct notations for large recursive ordinals eventually leading to and beyond a recursive ordinal that captures the combinatorial strength of Zermelo-Frankel set theory (ZF\(^4\)) and thus is strong enough to prove its consistency.

---

1Induction 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 \(x + 1\).

\[
\left[p(0) \land \forall x \in \mathbb{N} \left(p(x) \rightarrow p(x + 1)\right)\right] \rightarrow \forall x \in \mathbb{N} p(x)
\]

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

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

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

3A recursive ordinal, \(\alpha\), is one for which there exists a recursive notation. This a recursive process that enumerates notations for all ordinals \(\leq \alpha\) and a recursive process that decides the relative size of any two of these notations. Larger countable ordinals cannot have a recursive notation, but they can be defined as properties of recursive processes that operate on a partially enumerable domain. For more about this see Section 8.

4ZF is the widely used Zermelo-Frankel formulation of set theory. It can be thought of as a one page
Computers can help to deal with the inevitable complexity of strong systems of notations. They allow experiments to tests ones intuition. Thus a system of notations implemented in a computer program may be able to progress significantly beyond what is possible with pencil and paper alone.

2 Ordinal notations

The ordinals whose structure can be enumerated by an ideal computer program are called recursive. The smallest ordinal that is not recursive is the Church-Kleene ordinal $\omega_{CK}$. For simplicity this is written as $\omega_1$. Here the focus is on notations for recursive ordinals and larger countable ordinals in later sections. The set that defines a specific ordinal in set theory is unique, but there are many different computer programs that can define a notation system for the same recursive ordinal.

A notation system for a subset of the 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 it is desirable that an ordinal notation satisfy the following requirements:

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

2. There is an algorithm (or computer program) that can determine for every two ordinal notations $a$ and $b$ if $a < b$ or $a > b$ or $a = b$. 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 for a limit ordinal, $a$, will output an infinite sequence of ordinal notations, $b_i < a$ for all integers $i$. These outputs must satisfy the property that eventually a notation for 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. 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.

\begin{itemize}
\item $\omega_1$ 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 (except indirectly in describing an existing approach to ordinal collapsing in Section 8.5) we can simplify the notation for $\omega_{CK}$ to $\omega_1$.
\item 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.
\item To prove a property is true for some ordinal, $a$, one must prove the following.
\end{itemize}
Set theoretical approaches to the ordinals can mask the combinatorial structure that the ordinals implicitly define. This can be a big advantage in simplifying proofs, but it is only through the explicit development of that combinatorial structure that one can fully understand the ordinals. That understanding may be crucial to expanding the ordinal hierarchy. Beyond a certain point this is not practical without using computers as a research tool.

2.1 Ordinal functions and fixed points

Notations for ordinals are usually defined with strictly increasing ordinal functions on the countable ordinals. A fixed point of a function $f$ is an ordinal $a$ with $f(a) = a$. For example $f(x) = n + \alpha$ has every limit ordinal, $\alpha$, as a fixed point with $n$ an integer. In contrast $f(x) = \alpha + n$ is not a fixed point.

The Veblen hierarchy of ordinal notations is constructed by starting with the function $\omega^x$. Using this function, new functions are defined as a sequence of fixed points of previous functions\[18, 14, 9\]. The first of these functions, $\varphi(1, \alpha)$ enumerates the fixed points of $\omega^\alpha$. $\varphi(1, \alpha) = \varepsilon_\alpha$.

The range and domain of these functions are an expandable collection of ordinal notations defined by C++ class, Ordinal. The computational analog of fixed points in set theory involves the extension of an existing notation system. A fixed point can be thought of as representing the union of all notations that can be obtained by finite combinations of existing operations on existing ordinal notations. To represent this fixed point ordinal, the notation system must be expanded to include a new symbol for this ordinal. In addition the algorithms that operate on notations must be expanded to handle the additional symbol. In particular the recursive process that satisfies item 3 on page 8 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 $\omega$ the ordinal for the integers. It cannot be reached by any finite sequence of integer additions. Starting with a finite integer and adding $\omega$ to it cannot get past $\omega$. $n + \omega = \omega$ for all finite integers $n$.\[8\] The first fixed point for the function, $\omega^x$, is the ordinal $\varepsilon_0 = \omega + \omega^\omega + \omega^{(\omega^\omega)} + \omega^{(\omega(\omega^\omega))} + \ldots$. (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.) $\varepsilon_0$ represents the union of the ordinals represented by notations that can be obtained from finite sequences of operations starting with notations for the ordinals 0 and $\omega$ and the ordinal notation operations of successor (+1), addition, multiplication and exponentiation.

---

1. It is true of 0.

2. If it is true for any ordinal $b < a$ it must be true of the successor of $b$ or $b + 1$.

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

---

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

The class Ordinal is not restricted to notations for recursive ordinals. Admissible ordinals extend the concept of recursive ordinals by considering ordinal notations defined by Turing Machines (TM) with oracles\(^9\)\(^{12}\). There is an alternative way to extend the idea of recursive notations for ordinals. The recursive ordinals can be characterized by recursive processes that map integers to either processes like themselves or integers. That is a computer program that accepts an integer as input and outputs either a computer program like itself or an integer.

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

A recursive process satisfying the requirements on page 8 is well founded and represents the structure of a recursive ordinal. It has been shown that the concept of recursive process well founded for infinite sequences of integers can fully characterize the recursive ordinals\[^{16}\]. One can generalize this idea to recursive processes well founded for an infinite sequences of notations for recursive ordinals. And of course one can iterate this definition in simple and complex ways. In this way one can define countable ordinals \(\omega_1^{CK}\) as properties of recursive processes. In contrast to recursive ordinal notations, notations for larger ordinals cannot be associated with algorithms that fully enumerate the structure of the ordinal they represent. However it is possible to define a recursive function that in some ways serves as a substitute (see Section 8.3 on limitOrd).

2.3 Uncountable ordinals

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

Building incompleteness into the system from the ground up in lieu of claiming cardinality has an absolute meaning will, I believe, lead to a more powerful and more understandable mathematics. That is part of the reason I suspect a computational approach may extend

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

\(^{10}\)In mathematics a well founded relationship is one with no infinite descending chains. Any collection of objects ordered by the relationship must have a minimal element.

\(^{11}\)The formulation of this property requires quantification over the reals and thus is considered impredicative. (Impredicative sets have definitions that assume their own existence. Some reals can only be defined by quantifying over the set of all reals and this makes them questionable for some mathematicians.) In a computational approach one need not assume there is a set of all objects satisfying the property. Instead one can regard it as a computationally useful property and build objects that satisfy it in an expanding hierarchy. Impredicativity is replaced with explicit incompleteness.
the ordinal hierarchy significantly beyond what is possible by conventional proofs. Writing programs that define ordinal notations and operations on them provides powerful tools to help deal with combinatorial complexity that may vastly exceed the limits of what can be pursued with unaided human intelligence. This is true in most scientific fields and there is no reason to think that the foundations of mathematics is an exception. The core of this paper is about C++ code that defines a notation system for an initial fragment of the recursive ordinals and a subset of larger countable ordinals.

The development of this computational approach initially parallels the conventional approach through the Veblen hierarchy (described in Section 5). The C++ class Ordinal is incompletely specified. C++ subclasses and virtual functions allow a continued expansion of the class as described in the next section. The structure of the recursive ordinals defined in the process are fully specified. Ordinals ≥ ω1CK are partially specified.

Mathematics is an inherently creative activity. God did not create the integers, they like every other infinite set, are a human conceptual creation designed to abstract and generalize the real finite operations that God, or at least the forces of nature, did create. The anthology, God Created the Integers [10], collects the major papers in the history of mathematics. From these and the commentary it becomes clear that accepting any infinite totalities easily leads one to something like ZF. If the integers are a completed totality, then the rationals are ordered pairs of integers with 1 as their only common denominator. Reals are partitions of the rationals into those less than a given real and those greater than that real. This is the Dedekind cut. With the acceptance of that, one is well on the way to the power set axiom and wondering about the continuum hypothesis12 This mathematics is not false or even irrelevant, but it is only meaningful relative to a particular formal system. Thinking such questions are absolute leads one down a primrose path of pursuing as absolute what is not and cannot be objective truth beyond the confines of a particular formal system.

3 Program structure

The C++ programming language13 has two related features, subclasses and virtual functions that are useful in developing ordinal notations. The base class Ordinal, is an open ended programming structure that can be expanded with subclasses. It does not represent a specific set of ordinals and it is not limited to notations for recursive ordinals. Any function that takes Ordinal as an argument must allow any subclass of Ordinal to be passed to it as an argument.

12The continuum hypothesis is the assertion that the reals have the smallest cardinality greater than the cardinality of the integers. This means that if the reals can be mapped onto a set (with a unique real for every object in the set) and the the integers cannot then that set can be mapped onto the reals with a unique object for every real.

13C++ 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 a member function definition refer to the particular instance of the class from which the function is called.
3.1 virtual functions and subclasses

Virtual functions facilitate the expansion of the base class. For example there is a base class virtual member function compare that returns 1, 0, or -1 if its argument (which must be an element of class Ordinal) is less than, equal to or greater than the ordinal notation it is a member function of. The base class defines notations for ordinals less than $\varepsilon_0$. As the base class is expanded, an extension of the compare virtual function must be written to take care of cases involving ordinals greater than $\varepsilon_0$. Programs that call the compare function will continue to work properly. The correct version of compare will automatically be invoked depending on the type of the object (ordinal notation) from which compare is called. The original compare does not need to be changed but it does need to be written with the expansion in mind. If the argument to compare is not in the base class then the expanded function must be called. This can be done by calling compare recursively using the argument to the original call to compare as the class instance from which compare is called.

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

3.2 Ordinal normal forms

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

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

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

The $\alpha_k$ are ordinal notations and the $n_k$ are integers $> 0$.

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

The Ordinal base class represents an ordinal as a linked list of terms of the form $\omega^{\alpha_k}n_k$. This limits the base class to ordinals of the form $\omega^\alpha$ where $\alpha$ is a previously defined member of the base class. These are the ordinals less than $\varepsilon_0$. For larger ordinals we must define subclasses. The base class for each term of an Ordinal is CantorNormalElement and again subclasses are required to represent ordinals larger than or equal to $\varepsilon_0$.

\[^{14}\text{In the ordinal calculator } \varphi(1,\alpha) \text{ is displayed as } \varepsilon_\alpha.\]
3.3 Memory management

Most Ordinals are constructed from previously defined ones. Those constructions need to reference the ordinals used in defining them. These must not be deleted while the object that uses them still exists. This usually requires that Ordinals be created using the new C++ construct. Objects declared without using new are automatically deleted when the block in which they occur exits. This program does not currently implement any garbage collection. Ordinals created with new are not deleted until the program exits or the user explicitly deletes them. With the size of memory today this is usually not a problem but eventually a new version of the program should free space no longer being used.

4 Ordinal base class

All ordinal notations defined now or in the future in this system are include in base class, Ordinal. With no subclasses only notations for ordinals < ε₀ are defined. Sections 6, 7, 9 and 11 describe subclasses that extend the notation system to large recursive ordinals and to countable admissible ordinals.

The finite Ordinals and ω are defined using the constructor for class Ordinal. Other ordinals < ε₀ are usually defined with addition, multiplication and exponentiation of previously defined Ordinals (see Section 4.4).

The Ordinal for 12 is written as “const Ordinal& twelve = * new Ordinal(12)” in C++. The Ordinals for zero, one and omega are defined in global name space ord. The Ordinals two through six are defined as members of class Ordinal.

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

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

---

13Ordinal when capitalized and in tty font, refers to the expandable C++ class of ordinal notations.
14The constructor of a class object is a special member function that creates an instance of the class based on the its parameters.
15The integer ordinals are not defined in this program using the C++ int data type but a locally defined Int data type that uses the Mpir offshoot of the Gnu Multiple Precision Arithmetic Library to allow for arbitrarily large integers depending on the memory of the computer the program is run on.
16In the interactive ordinal calculator (see Appendix B) write “twelve = 12” or just use 12 in an expression.
17All 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.
18Reference to members of class Ordinal must include the prefix “Ordinal::” except in member functions of Ordinal.
19The interactive mode of entering ordinal expressions (see Appendix B) has the desired precedence and does not require parenthesis to perform exponentiation before multiplication.
4.1 normalForm and texNormalForm member functions

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

4.2 compare member function

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

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

4.3 limitElement member function

Ordinal member function limitElement has a single integer parameter. It is only defined for notations of limit ordinals and will abort if called from a successor Ordinal. Larger values of this argument produce larger ordinal notations as output. The union of the ordinals represented by the outputs for all integer inputs is equal to the ordinal represented by the Ordinal instance limitElement is called from. This function satisfies requirement 3 on page 8.

In the following description mathematical notation is mixed with C++ code. Thus limitElement(i) called from an Ordinal class instance that represents $\omega^\alpha$ is written.
This routine has a single exit code, ‘C’.
See Section 6.2 for more about this.

\[ \alpha = \sum_{m=0,1,...,k} \omega^{\alpha_m} n_m \] from equation 1
\[ \gamma = \sum_{m=0,1,...,k-1} \omega^{\alpha_m} n_m + \omega^{\alpha_k} (n_k - 1) \]

<table>
<thead>
<tr>
<th>Last Term ((\omega^{\alpha_k} n_k)) Condition</th>
<th>(\alpha).limitElement(i)</th>
</tr>
</thead>
<tbody>
<tr>
<td>(\alpha_k = 0) ((\alpha) is not a limit)</td>
<td>undefined abort</td>
</tr>
<tr>
<td>(\alpha_k = 1)</td>
<td>(\gamma + i)</td>
</tr>
<tr>
<td>(\alpha_k &gt; 1 \land \alpha_k) is a successor</td>
<td>(\gamma + \omega^{\alpha_k-1} i)</td>
</tr>
<tr>
<td>(\alpha_k) is a limit</td>
<td>(\gamma + \omega^{(\alpha_k).limitElement(i)})</td>
</tr>
</tbody>
</table>

Table 2: Cases for computing \(\text{Ordinal} :: \text{limitElement}\)

<table>
<thead>
<tr>
<th>Ordinal</th>
<th>1</th>
<th>2</th>
<th>10</th>
<th>100</th>
<th>786</th>
</tr>
</thead>
<tbody>
<tr>
<td>(\omega)</td>
<td>(\omega + 1)</td>
<td>(\omega + 2)</td>
<td>(\omega + 10)</td>
<td>(\omega + 100)</td>
<td>(\omega + 786)</td>
</tr>
<tr>
<td>(\omega^2)</td>
<td>(\omega)</td>
<td>(\omega^2)</td>
<td>(\omega^20)</td>
<td>(\omega^2100)</td>
<td>(\omega^2786)</td>
</tr>
<tr>
<td>(\omega^3)</td>
<td>(\omega^2)</td>
<td>(\omega^32)</td>
<td>(\omega^310)</td>
<td>(\omega^3100)</td>
<td>(\omega^3786)</td>
</tr>
<tr>
<td>(\omega^{\omega+2})</td>
<td>(\omega^{\omega+1})</td>
<td>(\omega^{\omega+12})</td>
<td>(\omega^{\omega+110})</td>
<td>(\omega^{\omega+1786})</td>
<td></td>
</tr>
<tr>
<td>(\omega^{\omega^{\omega+2}})</td>
<td>(\omega^{\omega^{\omega+1}})</td>
<td>(\omega^{\omega^{\omega+12}})</td>
<td>(\omega^{\omega^{\omega+110}})</td>
<td>(\omega^{\omega^{\omega+1786}})</td>
<td></td>
</tr>
</tbody>
</table>

Table 3: \(\text{Ordinal} :: \text{limitElement}\) examples.

as \((\omega^\alpha)\).limitElement(i).

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

### 4.4 Operators

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

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

\(^{22}\) is used for ‘exclusive or’ in C++ and has lower precedence than any arithmetic operator such as ‘+’. Thus C++ will evaluate \(x^y + 1\) as \(x^{(y+1)}\). Use parenthesis to override this as in \((x^y) + 1\).
<table>
<thead>
<tr>
<th>Operation</th>
<th>Example</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>addition</td>
<td>$\alpha + \beta$</td>
<td>add 1 to $\alpha \beta$ times</td>
</tr>
<tr>
<td>multiplication</td>
<td>$\alpha \times \beta$</td>
<td>add $\alpha \beta$ times</td>
</tr>
<tr>
<td>exponentiation</td>
<td>$\alpha^\beta$</td>
<td>multiply $\alpha \beta$ times</td>
</tr>
<tr>
<td>nested exponentiation</td>
<td>$\alpha^{\beta^\gamma}$</td>
<td>multiply $\alpha \beta^\gamma$ times</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

Table 4: Base class Ordinal operators.

<table>
<thead>
<tr>
<th>Expression</th>
<th>Cantor Normal Form</th>
<th>C++ code</th>
</tr>
</thead>
<tbody>
<tr>
<td>$(\omega^4 + 12)\omega$</td>
<td>$\omega^2$</td>
<td>$(\text{omega}^4+12)\text{omega}$</td>
</tr>
<tr>
<td>$\omega(\omega^4 + 12)$</td>
<td>$\omega^24 + \omega12$</td>
<td>$\text{omega}*(\text{omega}^4+12)$</td>
</tr>
<tr>
<td>$\omega(\omega^4 + 12)$</td>
<td>$\omega^2 + \omega^4$</td>
<td>$\omega^2 + \omega^4$</td>
</tr>
<tr>
<td>$(\omega + 4)(\omega + 5)$</td>
<td>$\omega^2 + \omega + 4$</td>
<td>$(\omega+4) \times (\omega+5)$</td>
</tr>
<tr>
<td>$(\omega + 2)^{\omega+2}$</td>
<td>$\omega^{\omega+2} + \omega^{\omega+1} + \omega^2$</td>
<td>$(\omega+2)^{\omega+2}$</td>
</tr>
<tr>
<td>$(\omega + 3)^{\omega+3}$</td>
<td>$\omega^{\omega+3} + \omega^{\omega+2} + \omega^{\omega+1} + \omega^3$</td>
<td>$(\omega+3)^{\omega+3}$</td>
</tr>
</tbody>
</table>

Table 5: Ordinal arithmetic examples

### 4.4.1 Addition

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

$$\alpha, \beta \text{ ordinals} \land \alpha \geq \beta \rightarrow \beta + \alpha \times \omega = \alpha \times \omega$$  \hspace{1cm} (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.

### 4.4.2 multiplication

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

$$(\omega + 3) \times 2 = (\omega + 3) + (\omega + 3) = \omega \times 2 + 3$$  \hspace{1cm} (3)

Like addition, multiplication works with the leading (most significant) terms of each operand in sequence. The operation that takes the product of terms is a member function of base class CantorNormalElement. It can be overridden by subclasses without affecting the algorithm than scans the terms of the two operands. When a subclass of Ordinal is added a subclass of CantorNormalElement must also be added.
CantorNormalElement and each of its subclasses is assigned a codeLevel that grows with the depth of class nesting. codeLevel for a CantorNormalElement is cantorCodeLevel. Any Cantor normal form term that is of the form $\omega^\alpha$ will be at this level regardless of the level of the terms of $\alpha$. codeLevel determines when a higher level function needs to be invoked. For example if we multiply $\alpha$ at cantorCodeLevel by $\beta$ at a higher level then a higher level routine must be used. This is accomplished by calling $\beta$.multiplyBy($\alpha$) which will invoke the virtual function multiplyBy in the subclass $\beta$ is an instance of.

The routine that multiplies two terms or CantorNormalElements 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 $\alpha_i$ and the factors are the $n_i$ in Equation 1. A CantorNormalElement with the computed exponent and factor is returned. If the exponents contain terms higher then cantorCodeLevel, this will be dealt with by the routine that does the addition of exponents.

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

### 4.4.3 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. 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

\footnote{A large integer exponent can require more memory than is available to store the result and abort the program.
<table>
<thead>
<tr>
<th>( \alpha )</th>
<th>( \beta )</th>
<th>( \alpha \times \beta )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \omega + 1 )</td>
<td>( \omega + 1 )</td>
<td>( \omega^2 + \omega + 1 )</td>
</tr>
<tr>
<td>( \omega + 1 )</td>
<td>( \omega + 2 )</td>
<td>( \omega^2 + \omega 2 + 1 )</td>
</tr>
<tr>
<td>( \omega + 1 )</td>
<td>( \omega^3 )</td>
<td>( \omega^4 )</td>
</tr>
<tr>
<td>( \omega + 1 )</td>
<td>( \omega^3 2 + 2 )</td>
<td>( \omega^4 2 + \omega 2 + 1 )</td>
</tr>
<tr>
<td>( \omega + 1 )</td>
<td>( \omega^4 + \omega^3 + \omega 7 + 3 )</td>
<td>( \omega^5 + \omega^4 + \omega^2 7 + \omega 3 + 1 )</td>
</tr>
<tr>
<td>( \omega + 1 )</td>
<td>( \omega^\omega 3 )</td>
<td>( \omega^\omega 3 )</td>
</tr>
<tr>
<td>( \omega + 2 )</td>
<td>( \omega + 1 )</td>
<td>( \omega^2 + \omega + 2 )</td>
</tr>
<tr>
<td>( \omega + 2 )</td>
<td>( \omega^2 + \omega + 2 )</td>
<td>( \omega^2 + \omega 2 + 2 )</td>
</tr>
<tr>
<td>( \omega + 2 )</td>
<td>( \omega^3 )</td>
<td>( \omega^4 )</td>
</tr>
<tr>
<td>( \omega + 2 )</td>
<td>( \omega^3 2 + 2 )</td>
<td>( \omega^4 2 + \omega 2 + 2 )</td>
</tr>
<tr>
<td>( \omega + 2 )</td>
<td>( \omega^4 + \omega^3 + \omega 7 + 3 )</td>
<td>( \omega^5 + \omega^4 + \omega^2 7 + \omega 3 + 2 )</td>
</tr>
<tr>
<td>( \omega^3 )</td>
<td>( \omega^\omega 3 )</td>
<td>( \omega^\omega 3 )</td>
</tr>
<tr>
<td>( \omega^3 )</td>
<td>( \omega + 1 )</td>
<td>( \omega^4 + \omega^3 2 + 2 )</td>
</tr>
<tr>
<td>( \omega^3 )</td>
<td>( \omega^2 + \omega + 2 )</td>
<td>( \omega^4 + \omega^3 4 + 2 )</td>
</tr>
<tr>
<td>( \omega^3 )</td>
<td>( \omega^3 )</td>
<td>( \omega^6 )</td>
</tr>
<tr>
<td>( \omega^3 2 + 2 )</td>
<td>( \omega^3 2 + 2 )</td>
<td>( \omega^6 2 + \omega^3 4 + 2 )</td>
</tr>
<tr>
<td>( \omega^3 2 + 2 )</td>
<td>( \omega^4 + \omega^3 + \omega 7 + 3 )</td>
<td>( \omega^7 + \omega^6 + \omega^4 7 + \omega^3 6 + 2 )</td>
</tr>
<tr>
<td>( \omega^3 2 + 2 )</td>
<td>( \omega^3 2 + 2 )</td>
<td>( \omega^7 + \omega^6 + \omega^4 7 + \omega^3 6 + 2 )</td>
</tr>
<tr>
<td>( \omega^3 2 + 2 )</td>
<td>( \omega^4 + \omega^3 + \omega 7 + 3 )</td>
<td>( \omega^8 + \omega^7 + \omega^5 7 + \omega^4 3 + \omega^3 + \omega 7 + 3 )</td>
</tr>
<tr>
<td>( \omega^3 2 + 2 )</td>
<td>( \omega^3 2 + 2 )</td>
<td>( \omega^\omega 3 )</td>
</tr>
<tr>
<td>( \omega^4 + \omega^3 + \omega 7 + 3 )</td>
<td>( \omega^4 + \omega^3 + \omega 7 + 3 )</td>
<td>( \omega^\omega 3 )</td>
</tr>
<tr>
<td>( \omega^4 + \omega^3 + \omega 7 + 3 )</td>
<td>( \omega^4 + \omega^3 + \omega 7 + 3 )</td>
<td>( \omega^\omega 3 )</td>
</tr>
<tr>
<td>( \omega^3 )</td>
<td>( \omega + 1 )</td>
<td>( \omega^\omega + 1 + \omega^\omega 3 )</td>
</tr>
<tr>
<td>( \omega^3 )</td>
<td>( \omega + 2 )</td>
<td>( \omega^\omega + 1 + \omega^\omega 6 )</td>
</tr>
<tr>
<td>( \omega^3 )</td>
<td>( \omega^3 )</td>
<td>( \omega^\omega + 3 2 + \omega^\omega 6 )</td>
</tr>
<tr>
<td>( \omega^3 )</td>
<td>( \omega^3 2 + 2 )</td>
<td>( \omega^\omega + 4 + \omega^\omega + 3 + \omega^\omega + 17 + \omega^\omega 9 )</td>
</tr>
<tr>
<td>( \omega^3 )</td>
<td>( \omega^4 + \omega^3 + \omega 7 + 3 )</td>
<td>( \omega^\omega + 2 3 )</td>
</tr>
</tbody>
</table>

Table 6: Ordinal multiply examples
The algorithm is to multiply the exponent of \( \omega \) from the first operand (the base) by the second operand. That product is used as the exponent of \( \omega \) in the term returned. Table 7 gives some examples with C++ code and some additional examples are shown in Table 8.

5 The Veblen hierarchy

This section gives a brief overview of the Veblen hierarchy and the \( \Delta \) operator. See [18, 14, 9] for a more a complete treatment. This is followed by the development of a computational approach for constructing notations for these ordinals up to (but not including) the large Veblen ordinal. We go further in Sections 8 and 9.

The Veblen hierarchy extends the recursive ordinals beyond \( \varepsilon_0 \). A Veblen hierarchy can be constructed from any strictly increasing continuous function \( f \), whose domain and range are the countable ordinals such that \( f(0) > 0 \). \( f(x) = \omega^x \) satisfies these conditions and is the starting point for constructing the standard Veblen hierarchy. One core idea is to define a new function from an existing one so that the new function enumerates the fixed points of the first one. A fixed point of \( f \) is a value \( v \) such that \( f(v) = v \). Given an infinite sequence of such functions one can define a new function that enumerates the common fixed points of all functions in the sequence. In this way one can iterate the construction of a new function up to any countable ordinal. The Veblen hierarchy based on \( f(x) = \omega^x \) is written as \( \varphi(\alpha, \beta) \) and defined as follows.

\[
\varphi(0, \beta) = \omega^\beta.
\]

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

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

From a Veblen hierarchy of the above sort, one can define a diagonalization function

---

\[ A \text{ continuous function, } f, \text{ on the ordinals must map limits to limits. Thus for every infinite limit ordinal } y, f(y) = \sup\{f(v) : v < y\}. \text{ A continuous strictly increasing function on the ordinals is called a normal function.} \]
\begin{center}
\begin{table}[h]
\begin{tabular}{|c|c|c|}
\hline
\(\alpha\) & \(\beta\) & \(\alpha^\beta\) \\
\hline
\(\omega + 1\) & \(\omega + 1\) & \(\omega^{\omega + 1} + 2\omega\) \\
\(\omega + 1\) & \(\omega + 2\) & \(\omega^{\omega + 2} + 2\omega + 1 + \omega^2\) \\
\(\omega + 1\) & \(\omega^3\) & \(\omega^{\omega^3}\) \\
\(\omega + 1\) & \(\omega^{3\omega + 2}\) & \(\omega^{\omega^{3\omega + 2} + 1} + \omega^3\) \\
\(\omega + 1\) & \(\omega^4 + 3\) & \(\omega^{\omega^4 + 3} + 2\omega + 3 + \omega^4\) \\
\(\omega + 1\) & \(\omega^\omega\) & \(\omega^{\omega^\omega}\) \\
\(\omega + 2\) & \(\omega + 1\) & \(\omega^{\omega + 1} + \omega^{\omega^2}\) \\
\(\omega + 2\) & \(\omega + 2\) & \(\omega^{\omega^2} + \omega^{\omega^2 + 12} + \omega^{\omega^3}\) \\
\(\omega + 2\) & \(\omega^3\) & \(\omega^{\omega^3}\) \\
\(\omega + 2\) & \(\omega^{3\omega + 2}\) & \(\omega^{\omega^{3\omega + 2} + 12} + \omega^{3\omega^2}\) \\
\(\omega + 2\) & \(\omega^4 + 3\) & \(\omega^{\omega^4 + 3} + 2\omega^{\omega^4 + 12} + \omega^{\omega^4}\) \\
\(\omega + 2\) & \(\omega^\omega\) & \(\omega^{\omega^\omega}\) \\
\(\omega^3 + 2\) & \(\omega + 1\) & \(\omega^{\omega + 1} + 2\omega^{\omega^3 + 2} + 2\omega + 3 + \omega^{\omega^3}\) \\
\(\omega^3 + 2\) & \(\omega + 2\) & \(\omega^{\omega^3 + 2} + \omega^{\omega^3 + 4} + \omega^{\omega^3\omega^2}\) \\
\(\omega^3 + 2\) & \(\omega^3\) & \(\omega^{\omega^3}\) \\
\(\omega^3 + 2\) & \(\omega^4 + 3\) & \(\omega^{\omega^3 + 2} + 4\omega + 3 + \omega^{\omega^3}\) \\
\(\omega^3 + 2\) & \(\omega^4 + 3\) & \(\omega^{\omega^3 + 2} + 3\omega + 4\omega + 3 + \omega^{\omega^3}\) \\
\hline
\end{tabular}
\caption{Ordinal exponential examples}
\end{table}
\end{center}
\( \varphi(x,0) \) from which a new Veblen hierarchy can be constructed. This can be iterated and the \( \Delta \) operator does this in a powerful way.

### 5.1 The delta operator

The \( \Delta \) operator is defined as follows\([14, 9]\).

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

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 \( \omega^x \). A generalized Veblen hierarchy is constructed by a similar iteration starting with any function on the countable ordinals, \( f(x) \), that is strictly increasing and continuous (see Note 24) with \( f(0) > 0 \). The \( \Delta \) 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 \( \Delta \) 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 classes in which the functions are defined and the functions themselves are designed to be extensible as future subclasses are added to the system.

### 5.2 A finite function hierarchy

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

\[
\varphi(\beta_1, \beta_2, \ldots, \beta_n).
\]

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 operand specifies the level of iteration of the next

\[^{25}\text{All references to ordinals in the context of describing the computational approach refer to ordinal notations. The word notation will sometimes be omitted when it is obviously meant and would be tedious to keep repeating.}\]
lowest level function. With a single parameter, Equation 4 is the function $\omega^x (\varphi(\alpha) = \omega^\alpha)$. With two parameters, $\varphi(\alpha, \beta)$, it is the Veblen hierarchy constructed from the base function $\omega^x$. With three parameters we have the $\Delta$ hierarchy built on this initial Veblen hierarchy. In particular the following holds.

$$\varphi(\alpha, \beta, x) = \Delta_\alpha \varphi_\beta^\alpha(x)$$

(5)

5.3 The finite function normal form

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

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

(6)

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

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

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

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

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

5.4 limitElement for finite functions

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

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
\[ \alpha = \varphi(\beta_1, \beta_2, \ldots, \beta_m) \text{ from Equation 6.} \]

lp1, rep1 and rep2 abbreviate limPlus_1 replace1 and replace2.

<table>
<thead>
<tr>
<th>Conditions on the least significant non zero parameters, leastOrd (index least) and nxtOrd (index nxt)</th>
<th>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() avoids fixed points by adding one to an ordinal with \text{psuedoCodeLevel} &gt; \text{cantorCodeLevel} (see Section 6.3 on fixed points). ret is the result returned.</th>
</tr>
</thead>
<tbody>
<tr>
<td>X</td>
<td>nxtOrd</td>
</tr>
<tr>
<td>FB</td>
<td>ignore</td>
</tr>
<tr>
<td>FD</td>
<td>successor</td>
</tr>
<tr>
<td>FL</td>
<td>ignore</td>
</tr>
<tr>
<td>FN</td>
<td>limit</td>
</tr>
</tbody>
</table>

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

Table 9: Cases for computing \( \varphi(\beta_1, \beta_2, \ldots, \beta_m).\text{limitElement(i)} \)
the result or output of the subroutine. Different sequences are generated based on the two least significant non zero parameters of $\varphi$ in Equation 4 and whether the least significant non zero term is the least significant term (including those that are zero). The idea is to construct an infinite sequence with a limit that is not reachable with a finite sequence of smaller notations.

5.5 An iterative functional hierarchy

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

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

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

$$\alpha_i = \varphi_{\gamma_i}(\beta_{1,i}, \beta_{2,i}, \ldots, \beta_{m,i})$$

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

$\gamma_i$, the subscript to $\varphi$, is the ordinal the functional hierarchy is iterated up to. $\varphi_0(\beta_1, \beta_2, \ldots, \beta_m) = \varphi(\beta_1, \beta_2, \ldots, \beta_n)$. $\varphi(0) = \varphi_1$ is the notation for an infinite union of the ordinals represented by finite functionals. Specifically it represents the union of ordinals with notations: $\varphi(1), \varphi(1,0), \varphi(1,0,0), \ldots$. $\varphi(1) = \omega$, $\varphi(1,0) = \epsilon_0$ and $\varphi(1,0,0) = \Gamma_0$. $\varphi_{\alpha}(\alpha) = \varphi(\alpha) = \omega^\alpha$ and $\varphi_{\gamma+1} = \varphi_{\gamma}(1) \cup \varphi_{\gamma}(1,0) \cup \varphi_{\gamma}(1,0,0), \ldots$.

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

6 FiniteFuncOrdinal class

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

The FiniteFuncOrdinal class should not be used directly to create ordinal notations. Instead use functions psi or finiteFunctional26. psi constructs notations for the initial Veblen hierarchy. It requires exactly two parameters (for the single parameter case use $\varphi(\alpha) = \omega^\alpha$). finiteFunctional accepts 3 to 5 parameters. For more than 5 use a

26In the interactive ordinal calculator the psi function can be use with any number of parameters to define a FiniteFuncOrdinal. See Section B.11.5 for some examples.
\[ \alpha = \varphi_\gamma(\beta_1, \beta_2, ..., \beta_m) \] from Equation 7. For this table \( m = 1 \).

<table>
<thead>
<tr>
<th>( X )</th>
<th>Condition</th>
<th>( \alpha ).limitElement(i)</th>
</tr>
</thead>
<tbody>
<tr>
<td>E</td>
<td>( \beta_1 = 0, \gamma ) limit</td>
<td>( \varphi_\gamma).limitElement(i).lp1(0)</td>
</tr>
<tr>
<td>F</td>
<td>( \beta_1 = 0, \gamma ) successor</td>
<td>( \varphi_{\gamma-1}(\varphi_{\gamma-1}(0), 0, 0, ..., 0) ) (i parameters)</td>
</tr>
<tr>
<td>G</td>
<td>( \beta_1 ) limit</td>
<td>( \varphi_\gamma(\beta_1).\text{limitElement}(i).lp1() )</td>
</tr>
<tr>
<td>H</td>
<td>( \beta_1 ) successor</td>
<td>( \varphi_\gamma).limitElement(i).lp1(( \varphi_\gamma(\beta_1 - 1).lp1(), 0, 0, ..., 0 )) (i parameters)</td>
</tr>
<tr>
<td>I</td>
<td>( \beta_1 ) successor</td>
<td>( \varphi_{\gamma-1}(\varphi_\gamma(\beta_1 - 1).lp1(), 0, 0, ..., 0) ) (i parameters)</td>
</tr>
<tr>
<td>J</td>
<td>( m &gt; 1 )</td>
<td>See Table 9 for more than one ( \beta_i ) parameter.</td>
</tr>
</tbody>
</table>

The ‘\( X \)’ column gives the exit code from \( \text{limitElement} \) (see Section 6.2). If \( \beta_1 = 0 \), it is the only \( \beta_i \). Leading zeros are normalized away.

Table 10: Cases for computing \( \varphi_\gamma(\beta_1, \beta_2, ..., \beta_m).\text{limitElement}(i) \)

\( \text{createParameters} \) can have 1 to 9 parameters all of which must be pointers to \( \text{Ordinals} \). Some examples are shown in Table 12. The direct use of the \( \text{Finite<FuncOrdinal} \) constructor is shown in Table 13. The ‘Ordinal’ column of both tables is created using \( \text{Ordinal}::\text{texNormalForm} \) which uses the standard notation for \( \varepsilon_\alpha \) and \( \Gamma_\alpha \) where appropriate. These functions reduce fixed points to their simplest expression and declare an \( \text{Ordinal} \) instead of a \( \text{Finite<FuncOrdinal} \) if appropriate. The first line of Table 12 is an example of this.

\( \text{Finite<FuncOrdinal} \) can be called with 3 or 4 parameters. For additional parameters, it can be called with a NULL terminated array of pointers to \( \text{Ordinals} \) notations. \( \text{createParameters} \) can be used to create this array as shown in the last line of Table 12.

6.1 compare member function

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

The \( \text{Finite<FuncNormalElement} \) object \( \text{compare} \) (or any \text{class} member function) is called from is ‘\text{this}’ in C++. The \( \text{CantorNormalElement} \) argument to \( \text{compare} \) is \( \text{trm} \). If \( \text{trm}.\text{codeLevel} > \text{finiteFuncCodeLevel} \) then \(-\text{trm}.\text{compare}(\text{\textthis}) \) is returned. This invokes the subclass member function associated with the subclass and \( \text{codeLevel} \) of \( \text{trm} \).

\( \text{CantorNormalElement}::\text{compare} \) only needs to test the exponents and if those are equal the factors of the two normal form terms being compared. An arbitrarily large number

---

27 The ‘&’ character in the last line in Table 12 is the C++ syntax that constructs a pointer to the object ‘&’ precedes.

28 \( \text{finiteFuncCodeLevel} \) is the \( \text{codeLevel} \) (see Section 4.4.2) of a \( \text{Finite<FuncNormalElement} \).
<table>
<thead>
<tr>
<th>Ordinal Calculator code</th>
<th>Ordinal</th>
</tr>
</thead>
<tbody>
<tr>
<td>( w^w )</td>
<td>( \omega^\omega )</td>
</tr>
<tr>
<td>epsilon( 0 )</td>
<td>( \varepsilon_0 )</td>
</tr>
<tr>
<td>epsilon( 1 )</td>
<td>( \varepsilon_1 )</td>
</tr>
<tr>
<td>psi( epsilon( 0 ), 1 )</td>
<td>( \varphi(\varepsilon_0, 1) )</td>
</tr>
<tr>
<td>epsilon( 2 )</td>
<td>( \varepsilon_2 )</td>
</tr>
<tr>
<td>gamma( 0 )</td>
<td>( \Gamma_0 )</td>
</tr>
<tr>
<td>psi( 1, 0, 1 )</td>
<td>( \varphi(1,0,1) )</td>
</tr>
<tr>
<td>psi( 1, 0, 2 )</td>
<td>( \varphi(1,0,2) )</td>
</tr>
<tr>
<td>gamma( 2 )</td>
<td>( \Gamma_2 )</td>
</tr>
<tr>
<td>psi( 1, 0, w )</td>
<td>( \varphi(1,0,\omega) )</td>
</tr>
<tr>
<td>psi( 1, epsilon( 0 ), w )</td>
<td>( \varphi(1,\varepsilon_0,\omega) )</td>
</tr>
<tr>
<td>psi( 1, 0, 0, 0 )</td>
<td>( \varphi(1,0,0,0) )</td>
</tr>
<tr>
<td>psi( 2, 0, 0 )</td>
<td>( \varphi(2,0,0) )</td>
</tr>
<tr>
<td>psi( 3, 0, 0 )</td>
<td>( \varphi(3,0,0) )</td>
</tr>
<tr>
<td>psi( w, 1, 0 )</td>
<td>( \varphi(\omega,1,0) )</td>
</tr>
<tr>
<td>psi( 1, 0, 0, 0 )</td>
<td>( \varphi(1,0,0,0) )</td>
</tr>
<tr>
<td>psi( w + 4, 0, 0 )</td>
<td>( \varphi(\omega+4,0,0) )</td>
</tr>
<tr>
<td>psi( 4, 12 )</td>
<td>( \varphi(4,12) )</td>
</tr>
<tr>
<td>psi( 3, ( w^2 ) + 2 )</td>
<td>( \varphi(3,\omega^2 + 2) )</td>
</tr>
<tr>
<td>psi( 1, 1, 1 )</td>
<td>( \varphi(1,1,1) )</td>
</tr>
<tr>
<td>psi( w, 1, 1 )</td>
<td>( \varphi(\omega,1,1) )</td>
</tr>
<tr>
<td>psi( 2, 0, 2, 1 )</td>
<td>( \varphi(2,0,2,1) )</td>
</tr>
<tr>
<td>psi( w, 0 )</td>
<td>( \varphi(\omega,0) )</td>
</tr>
<tr>
<td>psi( w, w )</td>
<td>( \varphi(\omega,\omega) )</td>
</tr>
<tr>
<td>psi( ( w^w ), 0, 0 )</td>
<td>( \varphi(\omega^\omega,0,0) )</td>
</tr>
<tr>
<td>psi( w, 0, 0 )</td>
<td>( \varphi(\omega,0,0) )</td>
</tr>
<tr>
<td>psi( w, 0, 0, 0 )</td>
<td>( \varphi(\omega,0,0,0) )</td>
</tr>
<tr>
<td>psi( 2, psi( 2, 0 ), epsilon( 0 ) )</td>
<td>( \varphi(2,\varphi(2,0),\varepsilon_0) )</td>
</tr>
<tr>
<td>psi( 3, psi( 2, 0 ), epsilon( 0 ) )</td>
<td>( \varphi(3,\varphi(2,0),\varepsilon_0) )</td>
</tr>
<tr>
<td>psi( 3, psi( 2, 0, 0, 0 ), w )</td>
<td>( \varphi(3,\varphi(2,0,0,0),\omega) )</td>
</tr>
<tr>
<td>psi( 3, psi( 2, 0, 0, 0 ), epsilon( 0 ) )</td>
<td>( \varphi(3,\varphi(2,0,0,0),\varepsilon_0) )</td>
</tr>
<tr>
<td>psi( 3, psi( 2, 0, 0, 0 ), epsilon( 1 ) )</td>
<td>( \varphi(3,\varphi(2,0,0,0),\varepsilon_1) )</td>
</tr>
<tr>
<td>psi( psi( 2, 0 ), gamma( 0 ), epsilon( 0 ) )</td>
<td>( \varphi(\varphi(2,0),\Gamma_0,\varepsilon_0) )</td>
</tr>
<tr>
<td>psi( w, 1 )</td>
<td>( \varphi(\omega,1) )</td>
</tr>
<tr>
<td>psi( w, 5 )</td>
<td>( \varphi(\omega,5) )</td>
</tr>
<tr>
<td>psi( w, 0, 1 )</td>
<td>( \varphi(\omega,0,1) )</td>
</tr>
</tbody>
</table>

Table 11: finiteFunctional Ordinal Calculator code examples
<table>
<thead>
<tr>
<th>C++ code</th>
<th>Ordinal</th>
</tr>
</thead>
<tbody>
<tr>
<td>psi(zero,omega)</td>
<td>$\omega$</td>
</tr>
<tr>
<td>psi(one,zero)</td>
<td>$\varepsilon_0$</td>
</tr>
<tr>
<td>psi(one,one)</td>
<td>$\varepsilon_1$</td>
</tr>
<tr>
<td>psi(eps0,one)</td>
<td>$\varphi(\varepsilon_0, 1)$</td>
</tr>
<tr>
<td>psi(one,Ordinal::two)</td>
<td>$\varepsilon_2$</td>
</tr>
<tr>
<td>finiteFunctional(one,zero,zero)</td>
<td>$\Gamma_0$</td>
</tr>
<tr>
<td>finiteFunctional(one,zero,one)</td>
<td>$\varphi(1, 0, 1)$</td>
</tr>
<tr>
<td>finiteFunctional(one,zero,Ordinal::two)</td>
<td>$\varphi(1, 0, 2)$</td>
</tr>
<tr>
<td>finiteFunctional(one,Ordinal::two,zero)</td>
<td>$\Gamma_2$</td>
</tr>
<tr>
<td>finiteFunctional(one,zero,omega)</td>
<td>$\varphi(1, 0, \omega)$</td>
</tr>
<tr>
<td>finiteFunctional(one,eps0,omega)</td>
<td>$\varphi(1, \varepsilon_0, \omega)$</td>
</tr>
<tr>
<td>finiteFunctional(one,zero,zero,zero)</td>
<td>$\varphi(1, 0, 0, 0)$</td>
</tr>
<tr>
<td>finiteFunctional(createParameters(&amp;one,&amp;zero,&amp;zero,&amp;zero,&amp;zero))</td>
<td>$\varphi(1, 0, 0, 0, 0)$</td>
</tr>
</tbody>
</table>

Table 12: finiteFunctional C++ code examples

<table>
<thead>
<tr>
<th>C++ code</th>
<th>Ordinal</th>
</tr>
</thead>
<tbody>
<tr>
<td>const Ordinal * const params[] =</td>
<td></td>
</tr>
<tr>
<td>{&amp;Ordinal::one,&amp;Ordinal::zero,0};</td>
<td></td>
</tr>
<tr>
<td>const FiniteFuncOrdinal eps0(params);</td>
<td>$\varepsilon_0$</td>
</tr>
<tr>
<td>Ordinal eps0_alt = psi(1,0);</td>
<td>$\varepsilon_0$</td>
</tr>
<tr>
<td>const FiniteFuncOrdinal eps0_alt2(1,0);</td>
<td>$\varepsilon_0$</td>
</tr>
<tr>
<td>const FiniteFuncOrdinal gamma0(1,0,0);</td>
<td>$\Gamma_0$</td>
</tr>
<tr>
<td>const FiniteFuncOrdinal gammaOmega(omega,0,0);</td>
<td>$\varphi(\omega, 0, 0)$</td>
</tr>
<tr>
<td>const FiniteFuncOrdinal gammx(gammaOmega,gamma0,omega);</td>
<td>$\varphi(\varphi(\omega, 0, 0), \Gamma_0, \omega)$</td>
</tr>
<tr>
<td>const FiniteFuncOrdinal big(1,0.0,0);</td>
<td>$\varphi(1,0,0,0)$</td>
</tr>
</tbody>
</table>

Table 13: FiniteFuncOrdinal C++ code examples
of Ordinal parameters are used to construct a FiniteFuncNormalElement. Thus a series of tests is required. This is facilitated by a member function CantorNormalElement::getMaxParameter that returns the largest parameter used in constructing this normal form term\(^{29}\). If trm.codeLevel < finiteFuncCodeLevel then trm > this only if the maximum parameter of trm is greater than this. However the value of factor for this must be ignored in making this comparison because \(trm \geq \omega^{\text{trm.getMaxParameter()}}\) and this will swamp the effect of any finite factor.

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

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

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

3. If this \(\leq\) the maximum parameter of trm return -1.

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

### 6.2 limitElement member function

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

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

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

---

\(^{29}\)For efficiency the constructor of a FiniteFuncNormalElement finds and saves the maximum parameter. For a CantorNormalElement the maximum parameter is the exponent as this is the only parameter that can be infinite. The case when the factor is larger than the exponent can be safely ignored.

\(^{30}\)OrdinalImpl is an internal implementation class that does most of the work for instances of an Ordinal.
Symbols used in this table and Table 18

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>lp1</td>
<td>limPlus_1 add a to a possible fixed point</td>
</tr>
<tr>
<td>le</td>
<td>limitElement</td>
</tr>
<tr>
<td>lNz</td>
<td>leastNonZero index of least significant nonzero ( \beta_i )</td>
</tr>
<tr>
<td>nlNz</td>
<td>nextLeastNonZero index of next to the least significant nonzero ( \beta_i )</td>
</tr>
<tr>
<td>rep1</td>
<td>replace1 replace 1 ( \beta_i ) parameter at specified index with specified value</td>
</tr>
<tr>
<td>rep2</td>
<td>replace2 replace 2 ( \beta_i ) parameters at specified indices with specified values</td>
</tr>
<tr>
<td>Rtn</td>
<td>return what is returned from code fragment</td>
</tr>
</tbody>
</table>

limitElement does all its work in limitElementCom or lower level routines. It has exit code AA when it calls cantorNormalElement(n) and exit code AB when it calls limitElementCom. See Section 5.4 for a description of exit codes in the X column.

\[ \alpha \text{ is from Expression 4, } \alpha = \varphi(\beta_1, \beta_2, ..., \beta_n). \]

<table>
<thead>
<tr>
<th>X</th>
<th>LimitTypeInfo</th>
<th>( \alpha \leq n(n) )</th>
</tr>
</thead>
<tbody>
<tr>
<td>FB</td>
<td>paramSuccZero</td>
<td>( b = \text{rp2}(lNz, \beta_{lNz} - 1, lNz + 1, 1); ) ( \text{for } i = 1; i &lt; n; i++ ) ( b = \text{rp2}(lNz, \beta_{lNz} - 1, lNz + 1, b); ) Rtn b</td>
</tr>
<tr>
<td>FD</td>
<td>paramsSucc</td>
<td>( b = \text{rep1}(sz - 1, \beta_{sz - 1} - 1); ) ( \text{for } i = 1; i &lt; n; i++ ) ( b = \text{rep2}(nNz, \beta_{nNz} - 1, nNz + 1, b.lp1); ) Rtn b</td>
</tr>
<tr>
<td>FL</td>
<td>paramLimit</td>
<td>( \text{rep1}(lNz, \beta_{lNz} \cdot l(n)) )</td>
</tr>
<tr>
<td>FN</td>
<td>paramNxtLimit</td>
<td>( b = \text{rep1}(sz - 1, \beta_{sz - 1} - 1); ) ( \text{rep2}(nNz, \beta_{nNz} \cdot l(n), nNz + 1, b.lp1) )</td>
</tr>
</tbody>
</table>

See Table 15 for examples

Table 14: FiniteFuncNormalElement::limitElementCom cases
<table>
<thead>
<tr>
<th>X</th>
<th>Ordinal</th>
<th>1</th>
<th>2</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>FB</td>
<td>$\varepsilon_0$</td>
<td>$\omega$</td>
<td>$\omega^\omega$</td>
<td>$\omega^\omega$</td>
</tr>
<tr>
<td>FB</td>
<td>$\varphi(2,0)$</td>
<td>$\varepsilon_1$</td>
<td>$\varepsilon_{\varepsilon_1+1}$</td>
<td>$\varepsilon_{\varepsilon_1+1}$</td>
</tr>
<tr>
<td>FB</td>
<td>$\varphi(5,0)$</td>
<td>$\varphi(4,1)$</td>
<td>$\varphi(4,\varphi(4,1)+1)$</td>
<td>$\varphi(4,\varphi(4,\varphi(4,1)+1)+1)$</td>
</tr>
<tr>
<td>FB</td>
<td>$\Gamma_0$</td>
<td>$\varepsilon_0$</td>
<td>$\varepsilon_{\varepsilon_0+1}$</td>
<td>$\varphi(\varepsilon_{\varepsilon_0+1})$</td>
</tr>
<tr>
<td>FB</td>
<td>$\Gamma_1$</td>
<td>$\varphi(1,0,1)$</td>
<td>$\varphi(1,0,\varphi(1,0,1)+1)$</td>
<td>$\varphi(1,0,\varphi(1,0,\varphi(1,0,1)+1)+1)$</td>
</tr>
<tr>
<td>FB</td>
<td>$\varphi(2,0,0)$</td>
<td>$\Gamma$</td>
<td>$\varphi(\Gamma_0+1,0,0)$</td>
<td>$\varphi(\varphi(\Gamma_0+1,0,0)+1,0,0)$</td>
</tr>
<tr>
<td>FB</td>
<td>$\varphi(3,0,0)$</td>
<td>$\varphi(2,1,0)$</td>
<td>$\varphi(2,\varepsilon_{\varepsilon_0+1})$</td>
<td>$\varphi(2,\varphi(2,\varepsilon_{\varepsilon_0+1})+1,0)$</td>
</tr>
<tr>
<td>FB</td>
<td>$\varphi(\omega,1,0)$</td>
<td>$\varphi(\omega,0,1)$</td>
<td>$\varphi(\omega,\varphi(\omega,0,1)+1)$</td>
<td>$\varphi(\omega,\varphi(\omega,\varphi(\omega,0,1)+1)+1)$</td>
</tr>
<tr>
<td>FB</td>
<td>$\varphi(1,0,0,0)$</td>
<td>$\Gamma_0$</td>
<td>$\varphi(\Gamma_0+1,0,0)$</td>
<td>$\varphi(\varphi(\Gamma_0+1,0,0)+1,0,0)$</td>
</tr>
<tr>
<td>FB</td>
<td>$\varphi(\omega+4,0,0)$</td>
<td>$\varphi(\omega+3,1,0)$</td>
<td>$\varphi(\omega,\varphi(\omega,\omega+3,1,0)+1)$</td>
<td>$\varphi(\omega,\varphi(\omega,\omega,\omega+3,1,0)+1)+1,0)$</td>
</tr>
<tr>
<td>FD</td>
<td>$\varphi(4,1,2)$</td>
<td>$\varphi(4,1,1)$</td>
<td>$\varphi(3,\varphi(4,1,1)+1)$</td>
<td>$\varphi(3,\varphi(3,\varphi(4,1,1)+1)+1)$</td>
</tr>
<tr>
<td>FD</td>
<td>$\varphi(3,\omega^2+2)$</td>
<td>$\varphi(3,\omega^2+1)$</td>
<td>$\varphi(2,\varphi(3,\omega^2+1)+1)$</td>
<td>$\varphi(2,\varphi(2,\varphi(3,\omega^2+1)+1)+1)$</td>
</tr>
<tr>
<td>FD</td>
<td>$\varphi(1,1,1)$</td>
<td>$\Gamma_1$</td>
<td>$\varphi(\omega,1,0)$</td>
<td>$\varphi(\omega,\varphi(\omega,\omega,0)+1)$</td>
</tr>
<tr>
<td>FD</td>
<td>$\varphi(2,0,2,1)$</td>
<td>$\varphi(2,0,2,0)$</td>
<td>$\varphi(2,0,1,\varphi(2,0,2,0)+1)$</td>
<td>$\varphi(2,0,1,\varphi(2,0,1,\varphi(2,0,2,0)+1)+1)$</td>
</tr>
<tr>
<td>FL</td>
<td>$\varphi(\omega,0)$</td>
<td>$\varepsilon_0$</td>
<td>$\varphi(\omega,2)$</td>
<td>$\varphi(\omega,3)$</td>
</tr>
<tr>
<td>FL</td>
<td>$\varphi(\omega,\omega)$</td>
<td>$\varphi(\omega,1)$</td>
<td>$\varphi(\omega,\omega)$</td>
<td>$\varphi(\omega^\omega,0,0)$</td>
</tr>
<tr>
<td>FL</td>
<td>$\varphi(\omega^\omega,0,0)$</td>
<td>$\varphi(\omega,0,0)$</td>
<td>$\varphi(\omega^\omega,0,0)$</td>
<td>$\varphi(\omega^\omega,0,0)$</td>
</tr>
<tr>
<td>FL</td>
<td>$\varphi(\omega,0,0)$</td>
<td>$\varphi(\omega,0,0)$</td>
<td>$\varphi(\omega,0,0)$</td>
<td>$\varphi(\omega,0,0)$</td>
</tr>
<tr>
<td>FL</td>
<td>$\varphi(2,\varphi(2,0),\varepsilon_0)$</td>
<td>$\varphi(2,\varphi(2,0),\omega)$</td>
<td>$\varphi(2,\varphi(2,0),\omega^\omega)$</td>
<td>$\varphi(2,\varphi(2,0),\omega^\omega)$</td>
</tr>
<tr>
<td>FL</td>
<td>$\varphi(3,\varphi(2,0),\varepsilon_0)$</td>
<td>$\varphi(3,\varphi(2,0),\omega)$</td>
<td>$\varphi(3,\varphi(2,0),\omega^\omega)$</td>
<td>$\varphi(3,\varphi(2,0),\omega^\omega)$</td>
</tr>
<tr>
<td>FL</td>
<td>$\varphi(3,\varphi(2,0,0,0),\omega)$</td>
<td>$\varphi(3,\varphi(2,0,0,0),\omega)$</td>
<td>$\varphi(3,\varphi(2,0,0,0),\omega^\omega)$</td>
<td>$\varphi(3,\varphi(2,0,0,0),\omega^\omega)$</td>
</tr>
<tr>
<td>FL</td>
<td>$\varphi(3,\varphi(2,0,0,0),\varepsilon_0)$</td>
<td>$\varphi(3,\varphi(2,0,0,0),\omega^\omega)$</td>
<td>$\varphi(3,\varphi(2,0,0,0),\omega^\omega)$</td>
<td>$\varphi(3,\varphi(2,0,0,0),\omega^\omega)$</td>
</tr>
<tr>
<td>FL</td>
<td>$\varphi(3,\varphi(2,0,0,0),\varepsilon_0)$</td>
<td>$\varphi(3,\varphi(2,0,0,0),\varepsilon_0+1)$</td>
<td>$\varphi(3,\varphi(2,0,0,0),\omega^\omega)$</td>
<td>$\varphi(3,\varphi(2,0,0,0),\omega^\omega)$</td>
</tr>
<tr>
<td>FL</td>
<td>$\varphi(3,\varphi(2,0,0,0),\varepsilon_0)$</td>
<td>$\varphi(3,\varphi(2,0,0,0),\varepsilon_0+1)$</td>
<td>$\varphi(3,\varphi(2,0,0,0),\omega^\omega)$</td>
<td>$\varphi(3,\varphi(2,0,0,0),\omega^\omega)$</td>
</tr>
<tr>
<td>FN</td>
<td>$\varphi(\omega,1)$</td>
<td>$\varphi(\omega,0)+1$</td>
<td>$\varphi(\omega,0)+1$</td>
<td>$\varphi(\omega,0)+1$</td>
</tr>
<tr>
<td>FN</td>
<td>$\varphi(\omega,5)$</td>
<td>$\varepsilon_{\varepsilon_0+1}$</td>
<td>$\varphi(\omega,4)+1$</td>
<td>$\varphi(\omega,4)+1$</td>
</tr>
<tr>
<td>FN</td>
<td>$\varphi(\omega,0,1)$</td>
<td>$\varphi(1,\varphi(\omega,0,0)+1,1)$</td>
<td>$\varphi(2,\varphi(\omega,0,0)+1,1)$</td>
<td>$\varphi(3,\varphi(\omega,0,0)+1,1)$</td>
</tr>
</tbody>
</table>

Table 15: FinitemfuncNormalElement::limitElementCom exit codes
6.3 fixedPoint member function

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

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

If the parameter selected has psuedoCodeLevel ≤ cantorCodeLevel, false is returned. If the maximum parameter psuedoCodeLevel > finiteFuncCodeLevel, true is returned. Otherwise a FiniteFuncOrdinal is constructed from all the parameters except that selected by the index which is set to zero31. If this value is less than the maximum parameter, true is returned and otherwise false.

6.4 operators

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

6.4.1 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 $\omega^x$, i.e. $a = \omega^a$. Thus the product of two such terms, $a$ and $b$ is $\omega^{a+b}$. Further the product of term $a$ at this level and $b = \omega^b$ for any term $b$ at cantorCodeLevel is $\omega^{a+b}$. Note in all cases if the first term

31If there is only one nonzero parameter (which must be the most significant), then the parameter array is increased by 1 and the most significant parameter is set to one in the value to compare with the maximum parameter.
has a factor other than 1 it will be ignored. The second term’s factor will be applied to the result.

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

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

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

Some examples are shown in Table 16.

---

\(^{32}\)A C++ static function is a member function of a class that is not associated with a particular instance of that class.
<table>
<thead>
<tr>
<th>$\alpha$</th>
<th>$\beta$</th>
<th>$\alpha \times \beta$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\varepsilon_0 + 2$</td>
<td>$\varepsilon_0 + 2$</td>
<td>$\omega^{(\varepsilon_0 + 2)} + (\varepsilon_0 + 2)$</td>
</tr>
<tr>
<td>$\varepsilon_0 + 2$</td>
<td>$\varepsilon_1 + \varepsilon_0 + 2$</td>
<td>$\omega^{(\varepsilon_0)} + (\varepsilon_0 + 2)$</td>
</tr>
<tr>
<td>$\varepsilon_0 + 2$</td>
<td>$\varphi(\omega^\omega, \omega + 3)$</td>
<td>$\varphi(\omega^\omega, \omega + 3)$</td>
</tr>
<tr>
<td>$\varepsilon_0 + 2$</td>
<td>$\varphi(3, \varepsilon_0, \varphi(\omega^\omega, \omega + 3))$</td>
<td>$\varphi(3, \varepsilon_0, \varphi(\omega^\omega, \omega + 3))$</td>
</tr>
<tr>
<td>$\varepsilon_1 + \varepsilon_0 + 2$</td>
<td>$\varepsilon_0 + 2$</td>
<td>$\omega^{(\varepsilon_1 + \varepsilon_0 + 2)} + (\varepsilon_0 + 2)$</td>
</tr>
<tr>
<td>$\varepsilon_1 + \varepsilon_0 + 2$</td>
<td>$\varphi(\omega^\omega, \omega + 3)$</td>
<td>$\varphi(\omega^\omega, \omega + 3)$</td>
</tr>
<tr>
<td>$\varepsilon_1 + \varepsilon_0 + 2$</td>
<td>$\varphi(3, \varepsilon_0, \varphi(\omega^\omega, \omega + 3))$</td>
<td>$\varphi(3, \varepsilon_0, \varphi(\omega^\omega, \omega + 3))$</td>
</tr>
<tr>
<td>$\varepsilon_1 + \varepsilon_0 + 2$</td>
<td>$\varphi(\omega^\omega, \omega + 3)$</td>
<td>$\varphi(\omega^\omega, \omega + 3)$</td>
</tr>
<tr>
<td>$\varepsilon_1 + \varepsilon_0 + 2$</td>
<td>$\varphi(3, \varepsilon_0, \varphi(\omega^\omega, \omega + 3))$</td>
<td>$\varphi(3, \varepsilon_0, \varphi(\omega^\omega, \omega + 3))$</td>
</tr>
<tr>
<td>$\omega^{(\varepsilon_0 + 2)}$</td>
<td>$\varphi(\omega^\omega, \omega + 3)$</td>
<td>$\varphi(\omega^\omega, \omega + 3)$</td>
</tr>
<tr>
<td>$\omega^{(\varepsilon_0 + 2)}$</td>
<td>$\varphi(3, \varepsilon_0, \varphi(\omega^\omega, \omega + 3))$</td>
<td>$\varphi(3, \varepsilon_0, \varphi(\omega^\omega, \omega + 3))$</td>
</tr>
<tr>
<td>$\omega^{(\varepsilon_0 + 2)}$</td>
<td>$\varphi(\omega^\omega, \omega + 3)$</td>
<td>$\varphi(\omega^\omega, \omega + 3)$</td>
</tr>
<tr>
<td>$\varphi(\omega^\omega, \omega + 3)$</td>
<td>$\varphi(3, \varepsilon_0, \varphi(\omega^\omega, \omega + 3))$</td>
<td>$\varphi(3, \varepsilon_0, \varphi(\omega^\omega, \omega + 3))$</td>
</tr>
<tr>
<td>$\varphi(\omega^\omega, \omega + 3)$</td>
<td>$\varphi(\omega^\omega, \omega + 3)$</td>
<td>$\varphi(\omega^\omega, \omega + 3)$</td>
</tr>
<tr>
<td>$\varphi(\omega^\omega, \omega + 3)$</td>
<td>$\varphi(3, \varepsilon_0, \varphi(\omega^\omega, \omega + 3))$</td>
<td>$\varphi(3, \varepsilon_0, \varphi(\omega^\omega, \omega + 3))$</td>
</tr>
<tr>
<td>$\varphi(\omega^\omega, \omega + 3)$</td>
<td>$\varphi(\omega^\omega, \omega + 3)$</td>
<td>$\varphi(\omega^\omega, \omega + 3)$</td>
</tr>
<tr>
<td>$\varphi(\omega^\omega, \omega + 3)$</td>
<td>$\varphi(3, \varepsilon_0, \varphi(\omega^\omega, \omega + 3))$</td>
<td>$\varphi(3, \varepsilon_0, \varphi(\omega^\omega, \omega + 3))$</td>
</tr>
<tr>
<td>$\varphi(\omega^\omega, \omega + 3)$</td>
<td>$\varphi(\omega^\omega, \omega + 3)$</td>
<td>$\varphi(\omega^\omega, \omega + 3)$</td>
</tr>
</tbody>
</table>

Table 16: FiniteFuncOrdinal multiply examples
6.4.2 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 $\omega^x$, i.e. $a = \omega^a$. The value of `factor` can be ignored in the base part of an exponential expression where the exponent is a limit ordinal. All infinite normal form terms are limit ordinals. Thus $a^b$ where both $a$ and $b$ are at `finiteFuncCodeLevel` and $b \leq a$ is $\omega^{ab}$ or equivalently $\omega^{\omega^{a+b}}$ which is the normal form result. If the base, $a$, of the exponential is at `cantorCodeLevel` then $a = \omega^a$ and the result is $\omega^{ab}$.

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

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

---

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

$^{34}$In C++ `boolean expression ? optionTrue : optionFalse` evaluates to `optionTrue` if `boolean expression` is true and `optionFalse` otherwise.
<table>
<thead>
<tr>
<th>$\alpha$</th>
<th>$\beta$</th>
<th>$\alpha^\beta$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\varepsilon_0 + 2$</td>
<td>$\varepsilon_0 + 2$</td>
<td>$\omega^{(\varepsilon_0^2)} + (\varepsilon_0 + 2)$</td>
</tr>
<tr>
<td>$\varepsilon_0 + 2$</td>
<td>$\varepsilon_1 + \varepsilon_0$</td>
<td>$\omega^{(\varepsilon_0^2)} + 3$</td>
</tr>
<tr>
<td>$\varepsilon_0 + 2$</td>
<td>$\varphi(\omega^{(\varepsilon_0^2)}, \omega + 3)$</td>
<td>$\varphi(\omega^{(\varepsilon_0^2)}, \omega + 3)$</td>
</tr>
<tr>
<td>$\varepsilon_0 + 2$</td>
<td>$\varphi(3, \varepsilon_0, \varphi(\omega^{(\varepsilon_0^2)}, \omega + 3))$</td>
<td>$\varphi(3, \varepsilon_0, \varphi(\omega^{(\varepsilon_0^2)}, \omega + 3))$</td>
</tr>
<tr>
<td>$\varepsilon_1 + \varepsilon_0$</td>
<td>$\varepsilon_0 + 2$</td>
<td>$\omega^{(\varepsilon_0^2)} + 3$</td>
</tr>
<tr>
<td>$\varepsilon_1 + \varepsilon_0$</td>
<td>$\varepsilon_1 + \varepsilon_0$</td>
<td>$\varphi(\omega^{(\varepsilon_0^2)}, \omega + 3)$</td>
</tr>
<tr>
<td>$\varepsilon_1 + \varepsilon_0$</td>
<td>$\varphi(3, \varepsilon_0, \varphi(\omega^{(\varepsilon_0^2)}, \omega + 3))$</td>
<td>$\varphi(3, \varepsilon_0, \varphi(\omega^{(\varepsilon_0^2)}, \omega + 3))$</td>
</tr>
<tr>
<td>$\omega^{(\varepsilon_0^2)} + 3$</td>
<td>$\varepsilon_1 + \varepsilon_0$</td>
<td>$\varepsilon_1 + \varepsilon_0$</td>
</tr>
<tr>
<td>$\omega^{(\varepsilon_0^2)} + 3$</td>
<td>$\varphi(\omega^{(\varepsilon_0^2)}, \omega + 3)$</td>
<td>$\varphi(\omega^{(\varepsilon_0^2)}, \omega + 3)$</td>
</tr>
<tr>
<td>$\omega^{(\varepsilon_0^2)} + 3$</td>
<td>$\varphi(3, \varepsilon_0, \varphi(\omega^{(\varepsilon_0^2)}, \omega + 3))$</td>
<td>$\varphi(3, \varepsilon_0, \varphi(\omega^{(\varepsilon_0^2)}, \omega + 3))$</td>
</tr>
<tr>
<td>$\varphi(\omega^{(\varepsilon_0^2)}, \omega + 3)$</td>
<td>$\varepsilon_0 + 2$</td>
<td>$\varepsilon_0 + 2$</td>
</tr>
<tr>
<td>$\varphi(\omega^{(\varepsilon_0^2)}, \omega + 3)$</td>
<td>$\varepsilon_1 + \varepsilon_0$</td>
<td>$\varepsilon_1 + \varepsilon_0$</td>
</tr>
<tr>
<td>$\varphi(\omega^{(\varepsilon_0^2)}, \omega + 3)$</td>
<td>$\omega^{(\varepsilon_0^2)} + 3$</td>
<td>$\varphi(3, \varepsilon_0, \varphi(\omega^{(\varepsilon_0^2)}, \omega + 3))$</td>
</tr>
<tr>
<td>$\varphi(\omega^{(\varepsilon_0^2)}, \omega + 3)$</td>
<td>$\varphi(\omega^{(\varepsilon_0^2)}, \omega + 3)$</td>
<td>$\varphi(\omega^{(\varepsilon_0^2)}, \omega + 3)$</td>
</tr>
<tr>
<td>$\varphi(3, \varepsilon_0, \varphi(\omega^{(\varepsilon_0^2)}, \omega + 3))$</td>
<td>$\varepsilon_0 + 2$</td>
<td>$\varphi(3, \varepsilon_0, \varphi(\omega^{(\varepsilon_0^2)}, \omega + 3))$</td>
</tr>
<tr>
<td>$\varphi(3, \varepsilon_0, \varphi(\omega^{(\varepsilon_0^2)}, \omega + 3))$</td>
<td>$\varepsilon_1 + \varepsilon_0$</td>
<td>$\varepsilon_1 + \varepsilon_0$</td>
</tr>
<tr>
<td>$\varphi(3, \varepsilon_0, \varphi(\omega^{(\varepsilon_0^2)}, \omega + 3))$</td>
<td>$\omega^{(\varepsilon_0^2)} + 3$</td>
<td>$\varphi(3, \varepsilon_0, \varphi(\omega^{(\varepsilon_0^2)}, \omega + 3))$</td>
</tr>
<tr>
<td>$\varphi(3, \varepsilon_0, \varphi(\omega^{(\varepsilon_0^2)}, \omega + 3))$</td>
<td>$\varphi(\omega^{(\varepsilon_0^2)}, \omega + 3)$</td>
<td>$\varphi(\omega^{(\varepsilon_0^2)}, \omega + 3)$</td>
</tr>
<tr>
<td>$\varphi(3, \varepsilon_0, \varphi(\omega^{(\varepsilon_0^2)}, \omega + 3))$</td>
<td>$\varphi(3, \varepsilon_0, \varphi(\omega^{(\varepsilon_0^2)}, \omega + 3))$</td>
<td>$\varphi(3, \varepsilon_0, \varphi(\omega^{(\varepsilon_0^2)}, \omega + 3))$</td>
</tr>
</tbody>
</table>

Table 17: FiniteFuncOrdinal exponential examples
See Table 14 for symbols used here.

<table>
<thead>
<tr>
<th>$\alpha$ is from Expression 4, $\alpha = \varphi(\beta_1, \beta_2, \ldots, \beta_n)$.</th>
</tr>
</thead>
<tbody>
<tr>
<td>$X$</td>
</tr>
<tr>
<td>OFA</td>
</tr>
<tr>
<td>OFB</td>
</tr>
</tbody>
</table>

See Table TO BE ADDED for examples

Table 18: FiniteFuncNormalElement::limitOrdCom

6.5 limitOrd member function

limitOrd is analogous to limitElement but it supports ordinals $\geq \omega_1^{CK}$, the ordinal of the recursive ordinals. We have do not have notations for such ordinals at the code level discusses up to this point. This routine will only be used in conjunction with notations defined in later sections. limitOrdCom is a subroutine called by limitOrd to do most of the work and structured in such a way that it can be used by notations at higher code level. It creates virtual objects that can automatically fill in parameters from higher level objects that call limitOrdCom.

7 IterFuncOrdinal class

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

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 $\gamma_i$ in Equation 7. The second gives a NULL terminated array of pointers to Ordinals which are the $\beta_{i,j}$ in Equation 7. This second parameter is optional and can be created with function createParameters described in Section 6. Some examples are shown in Table 19.

iterativeFunctional creates an Ordinal or FiniteFuncOrdinal instead of an IterFuncOrdinal if appropriate. The first three lines of Table 19 are examples. It also reduces fixed points to their simplest possible expression. The last line of the table is an example.

---

35The interactive ordinal calculator supports a TeX like syntax. To define $\varphi(a, b, c, d)$ write $\psi_{\{a\}}(b, c, d)$. Any number of parameters in parenthesis may be entered or the parenthesis may be omitted. See Section B.11.6 for examples.
‘cp’ stands for function `createParameters`

<table>
<thead>
<tr>
<th>C++ code</th>
<th>Ordinal</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>iterativeFunctional(zero,cp(&amp;one))</code></td>
<td>$\omega$</td>
</tr>
<tr>
<td><code>iterativeFunctional(zero,cp(&amp;one,&amp;zero))</code></td>
<td>$\epsilon_0$</td>
</tr>
<tr>
<td><code>iterativeFunctional(zero,cp(&amp;one,&amp;one,&amp;zero))</code></td>
<td>$\Gamma_1$</td>
</tr>
<tr>
<td><code>iterativeFunctional(one)</code></td>
<td>$\varphi_1(1)$</td>
</tr>
<tr>
<td><code>iterativeFunctional(one,cp(&amp;one))</code></td>
<td>$\varphi_1(1,\omega)$</td>
</tr>
<tr>
<td><code>iterativeFunctional(one,cp(&amp;one,&amp;omega))</code></td>
<td>$\varphi_1(1,\omega,0,0)$</td>
</tr>
<tr>
<td><code>iterativeFunctional(one,cp(&amp;one,&amp;omega,&amp;zero,&amp;zero))</code></td>
<td>$\varphi_1(1,\omega,0,0)$</td>
</tr>
<tr>
<td><code>iterativeFunctional(omega,cp(&amp;one,&amp;omega,&amp;zero,&amp;zero))</code></td>
<td>$\varphi_\omega(1,\omega,0,0)$</td>
</tr>
<tr>
<td><code>iterativeFunctional(omega)</code></td>
<td>$\varphi_\omega$</td>
</tr>
<tr>
<td><code>iterativeFunctional(one,cp(&amp;iterativeFunctional(omega)))</code></td>
<td></td>
</tr>
</tbody>
</table>

Table 19: `iterativeFunctional` C++ code examples

### 7.1 compare member function

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

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

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

If `trm.codeLevel < iterFuncCodeLevel` then `trm > this` only if the maximum parameter of `trm` is greater than `this`. However the values of `factor` must be ignored in making this comparison because `trm \geq \omega_{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 $\leq 0$, -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`) ($\gamma_i$ from Equation 7) of `this` is compared to the `functionLevel` of `trm`. If the result is nonzero it is returned.
If no result is obtained `IterFuncNormalElement::compareFiniteParams` is called to compare in sequence the number of parameters of the two terms and then the size of each argument in succession starting with the most significant. If any difference is encountered that is returned as the result. If not the difference in the factors of the two terms is returned.

### 7.2 limitElement member function

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

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

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

---

36 The 'X' column in Tables 9, 10 and others connects table entries to a section of code preceding a return. The return is implemented with a macro that has an exit code string as a parameter. If debugging mode is enabled for the appropriate functions these exit codes are displayed. For compare functions use `setDbg compare` and for `limitElement` or `limitOrd` related functions use `setDbg limit` in the ordinal calculator.

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

38
<table>
<thead>
<tr>
<th>X</th>
<th>Condition(s)</th>
<th>LimitTypeInfo</th>
<th>( \alpha .\text{le}(n) )</th>
</tr>
</thead>
<tbody>
<tr>
<td>IF</td>
<td>at least 1 ( \beta_i(i &gt; 1) ) is nonzero</td>
<td>paramSucc paramSuccZero paramsSucc paramLimit paramNxtLimit</td>
<td>FiniteFuncNormalElement::limitElementCom(n)</td>
</tr>
<tr>
<td>IG</td>
<td>( \gamma .\text{isSc} \land sz == 0; )</td>
<td>functionSucc</td>
<td>( b=\varphi_{\gamma-1}; ) ( \text{Rtn } \varphi_{\gamma-1}(b.lp1,0,...,0) ) (n-1 zeros)</td>
</tr>
<tr>
<td>II</td>
<td>( \gamma .\text{isSc} \land \beta_1.\text{isSc} \land sz == 1 )</td>
<td>functionNxtSucc</td>
<td>( b=\varphi_{\gamma}(\beta_1 - 1); ) ( \text{Rtn } \varphi_{\gamma-1}(b.lp1,0,...,0) ) (n-1 zeros)</td>
</tr>
<tr>
<td>IJ</td>
<td>( \gamma .\text{isLm} \land sz == 0 )</td>
<td>functionLimit</td>
<td>( \varphi_{\gamma.\text{le}(n)} )</td>
</tr>
<tr>
<td>IK</td>
<td>( \kappa .\text{isLm} \land sz == 1 \land \beta_1.\text{isSc} )</td>
<td>functionNxtLimit</td>
<td>( \varphi_{\gamma.\text{le}(n)}(\varphi_{\gamma}(\beta_0 - 1).lp1) )</td>
</tr>
</tbody>
</table>

See Table for examples

Table 20: IterFuncNormalElement::limitElementCom cases
A is an exit code (see Table 20). UpX is a higher level exit code from a calling routine.

<table>
<thead>
<tr>
<th>X</th>
<th>UpX</th>
<th>Ordinal</th>
<th>1</th>
<th>2</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>FB</td>
<td>IF</td>
<td>ϕω(1,0)</td>
<td>ϕω(1)</td>
<td>ϕω(ϕω(1) + 1)</td>
<td>ϕω(ϕω(ϕω(1) + 1) + 1)</td>
</tr>
<tr>
<td>FB</td>
<td>IF</td>
<td>ϕ1(1,0)</td>
<td>ϕ1(1)</td>
<td>ϕ1(ϕ1(1,0) + 1,0)</td>
<td>ϕ1(ϕ1(ϕ1(1,0) + 1,0) + 1)</td>
</tr>
<tr>
<td>FB</td>
<td>IF</td>
<td>ϕ3(1,0)</td>
<td>ϕ3(1)</td>
<td>ϕ3(ϕ3(1,0) + 1,0)</td>
<td>ϕ3(ϕ3(ϕ3(1,0) + 1,0) + 1)</td>
</tr>
<tr>
<td>FB</td>
<td>IF</td>
<td>ϕ1(ω,1,0)</td>
<td>ϕ1(ω,0,1)</td>
<td>ϕ1(ω,0,ϕ1(ω,0,1) + 1)</td>
<td>ϕ1(ω,0,ϕ1(ω,0,1) + 1) + 1)</td>
</tr>
<tr>
<td>FB</td>
<td>IF</td>
<td>ϕTα(1,0)</td>
<td>ϕTα(1)</td>
<td>ϕTα(ϕTα(1) + 1)</td>
<td>ϕTα(ϕTα(ϕTα(1) + 1) + 1)</td>
</tr>
<tr>
<td>FD</td>
<td>IF</td>
<td>ϕTα(ε0,1,1)</td>
<td>ϕTα(ε0,1,0)</td>
<td>ϕTα(ε0,0,ϕTα(ε0,1,0) + 1)</td>
<td>ϕTα(ε0,0,ϕTα(ε0,1,0) + 1) + 1)</td>
</tr>
<tr>
<td>FL</td>
<td>IF</td>
<td>ϕ12(ω,0)</td>
<td>ϕ12(1,0)</td>
<td>ϕ12(2,0)</td>
<td>ϕ12(3,0)</td>
</tr>
<tr>
<td>FL</td>
<td>IF</td>
<td>ϕ1(ε0)</td>
<td>ϕ1(ω)</td>
<td>ϕ1(ω^ω)</td>
<td>ϕ1(ω^ω^ω)</td>
</tr>
<tr>
<td>FL</td>
<td>IF</td>
<td>ϕTα(ε0)</td>
<td>ϕTα(ω)</td>
<td>ϕTα(ω^ω)</td>
<td>ϕTα(ω^ω^ω)</td>
</tr>
<tr>
<td>FL</td>
<td>IF</td>
<td>ϕTα(ε0,0,0)</td>
<td>ϕTα(ω,0)</td>
<td>ϕTα(ω^ω,0)</td>
<td>ϕTα(ω^ω^ω,0)</td>
</tr>
<tr>
<td>FN</td>
<td>IF</td>
<td>ϕ1(ω,1)</td>
<td>ϕ1(ω,0,1) + 1)</td>
<td>ϕ1(2,0,1)</td>
<td>ϕ1(3,0,1)</td>
</tr>
<tr>
<td>FN</td>
<td>IF</td>
<td>ϕTα(ε0,0)</td>
<td>ϕTα(ω,0)</td>
<td>ϕTα(ω^ω,0)</td>
<td>ϕTα(ω^ω^ω,0)</td>
</tr>
<tr>
<td>IG</td>
<td></td>
<td>ϕω+1</td>
<td>ϕω(ϕω + 1)</td>
<td>ϕω(ϕω + 1,0)</td>
<td>ϕω+1(ϕω + 1)</td>
</tr>
<tr>
<td>IG</td>
<td></td>
<td>ϕ2</td>
<td>ϕ1(ϕ1 + 1)</td>
<td>ϕ1(ϕ1 + 1,0)</td>
<td>ϕ1(ϕ1 + 1,0)</td>
</tr>
<tr>
<td>IG</td>
<td></td>
<td>ϕ3</td>
<td>ϕ2(ϕ2 + 1)</td>
<td>ϕ2(ϕ2 + 1,0)</td>
<td>ϕ2(ϕ2 + 1,0)</td>
</tr>
<tr>
<td>II</td>
<td></td>
<td>ϕ1(1)</td>
<td>ϕ1(1) + 1</td>
<td>ϕ1(1) + 1,0)</td>
<td>ϕ1(1) + 1,0)</td>
</tr>
<tr>
<td>II</td>
<td></td>
<td>ϕ2(2)</td>
<td>ϕ2(2,1) + 1</td>
<td>ϕ2(ϕ2(2) + 1,0)</td>
<td>ϕ2(ϕ2(2) + 1,0)</td>
</tr>
<tr>
<td>II</td>
<td></td>
<td>ϕ2(4)</td>
<td>ϕ2(3) + 1</td>
<td>ϕ2(ϕ2(3) + 1,0)</td>
<td>ϕ2(ϕ2(3) + 1,0)</td>
</tr>
<tr>
<td>II</td>
<td></td>
<td>ϕω+1(1)</td>
<td>ϕω+1(1)</td>
<td>ϕω(ϕω+1 + 1,0)</td>
<td>ϕω(ϕω+1 + 1,0)</td>
</tr>
<tr>
<td>LJ</td>
<td></td>
<td>ϕω</td>
<td>ϕ1</td>
<td>ϕ2</td>
<td>ϕ3</td>
</tr>
<tr>
<td>LJ</td>
<td>AB</td>
<td>ϕε0</td>
<td>ϕω</td>
<td>ϕω</td>
<td>ϕω</td>
</tr>
<tr>
<td>LJ</td>
<td>AB</td>
<td>ϕTα</td>
<td>ϕε0+1</td>
<td>ϕTα(ϕε0+1,0)+1</td>
<td>ϕTα(ϕε0+1,0)+1,0+1</td>
</tr>
<tr>
<td>LJ</td>
<td>AB</td>
<td>ϕTα(3,0,0)</td>
<td>ϕTα(2,1,0)+1</td>
<td>ϕTα(2,1,0)+1,0+1</td>
<td>ϕTα(2,1,0)+1,0+1+1</td>
</tr>
<tr>
<td>IK</td>
<td></td>
<td>ϕω(1)</td>
<td>ϕω(ϕω + 1)</td>
<td>ϕ2(ϕω + 1)</td>
<td>ϕ3(ϕω + 1)</td>
</tr>
<tr>
<td>IK</td>
<td>AB</td>
<td>ϕω(5)</td>
<td>ϕω(ϕω(4) + 1)</td>
<td>ϕω(ϕω(4) + 1)</td>
<td>ϕω(ϕω(4) + 1)</td>
</tr>
<tr>
<td>IK</td>
<td>AB</td>
<td>ϕTα(1)</td>
<td>ϕε0+1(ϕTα + 1)</td>
<td>ϕTα(ϕε0+1,0+1)</td>
<td>ϕTα(ϕε0+1,0+1)</td>
</tr>
</tbody>
</table>

Table 21: IterFuncNormalElement::limitElementCom exit codes
7.3 **fixedPoint member function**

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

- The function level, \( \gamma \).
- An index specifying the largest parameter of the ordinal notation being constructed. If the largest parameter is the function level the index has the value `iterMaxParam` defined as \(-1\) in an `enum`.
- The function parameters as an array of pointers to `Ordinals`. These are the \( \beta_j \) in a normal form term in Equation 7.

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

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

7.4 **operators**

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

Some multiply examples are shown in Table 22. Some exponential examples are shown in Table 23.
<table>
<thead>
<tr>
<th>$\alpha$</th>
<th>$\beta$</th>
<th>$\alpha \times \beta$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\varphi_1$</td>
<td>$\varphi_1$</td>
<td>$\omega(\varphi_1^2)$</td>
</tr>
<tr>
<td>$\varphi_1$</td>
<td>$\varphi_3$</td>
<td>$\varphi_3$</td>
</tr>
<tr>
<td>$\varphi_1$</td>
<td>$\varphi_3(1,0,0)$</td>
<td>$\varphi_3(1,0,0)$</td>
</tr>
<tr>
<td>$\varphi_1$</td>
<td>$\varphi_3(1,0,0)$</td>
<td>$\varphi_3(1,0,0)$ + $\omega(\varphi_1^2)$</td>
</tr>
<tr>
<td>$\varphi_1$</td>
<td>$\varphi_3(1,0,0)$</td>
<td>$\varphi_3(1,0,0)$ + $\omega(\varphi_1^2)$</td>
</tr>
<tr>
<td>$\varphi_1$</td>
<td>$\varphi_3(1,0,0)$ + $\varphi_1$</td>
<td>$\varphi_1(\omega, 1) + \omega(\varphi_1^2)$</td>
</tr>
<tr>
<td>$\varphi_1$</td>
<td>$\varphi_3(1,0,0)$ + $\varphi_1$</td>
<td>$\varphi_1(\omega, 1)$</td>
</tr>
<tr>
<td>$\varphi_1$</td>
<td>$\varphi_3(1,0,0)$ + $\varphi_1$</td>
<td>$\varphi_1(\omega, 1)$</td>
</tr>
<tr>
<td>$\varphi_1$</td>
<td>$\varphi_3(1,0,0)$ + $\varphi_1$</td>
<td>$\varphi_1(\omega, 1)$</td>
</tr>
<tr>
<td>$\varphi_0 + \varphi_1(\omega, 1, 0)$ + $\varphi_1$</td>
<td>$\varphi_2 + \varphi_1(\omega, 1, 0)$ + $\varphi_1$</td>
<td>$\varphi_2 + \varphi_1(\omega, 1, 0)$ + $\varphi_1$</td>
</tr>
<tr>
<td>$\varphi_0 + \varphi_1(\omega, 1, 0)$ + $\varphi_1$</td>
<td>$\varphi_2 + \varphi_1(\omega, 1, 0)$ + $\varphi_1$</td>
<td>$\varphi_2 + \varphi_1(\omega, 1, 0)$ + $\varphi_1$</td>
</tr>
<tr>
<td>$\varphi_0 + \varphi_1(\omega, 1, 0)$ + $\varphi_1$</td>
<td>$\varphi_2 + \varphi_1(\omega, 1, 0)$ + $\varphi_1$</td>
<td>$\varphi_2 + \varphi_1(\omega, 1, 0)$ + $\varphi_1$</td>
</tr>
<tr>
<td>$\varphi_0 + \varphi_1(\omega, 1, 0)$ + $\varphi_1$</td>
<td>$\varphi_2 + \varphi_1(\omega, 1, 0)$ + $\varphi_1$</td>
<td>$\varphi_2 + \varphi_1(\omega, 1, 0)$ + $\varphi_1$</td>
</tr>
</tbody>
</table>

Table 22: IterFuncOrdinal multiply examples
<table>
<thead>
<tr>
<th>$\alpha$</th>
<th>$\beta$</th>
<th>$\alpha^3$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\varphi_1$</td>
<td>$\varphi_1$</td>
<td>$\omega^3(\varphi_1)$</td>
</tr>
<tr>
<td>$\varphi_1$</td>
<td>$\varphi_3$</td>
<td>$\omega^3(\varphi_1)$</td>
</tr>
<tr>
<td>$\varphi_1$</td>
<td>$\varphi_3(1,0,0)$</td>
<td>$\omega^3(\varphi_1)$</td>
</tr>
<tr>
<td>$\varphi_1$</td>
<td>$\varphi_1(\omega,1)$</td>
<td>$\omega^3(\varphi_1)$</td>
</tr>
<tr>
<td>$\varphi_1$</td>
<td>$\varphi_1(\omega,1,0) + \varphi_1$</td>
<td>$\omega^3(\varphi_1)$</td>
</tr>
<tr>
<td>$\varphi_1$</td>
<td>$\varphi_{e0} + \varphi_1(\omega,1,0) + \varphi_1$</td>
<td>$\omega^3(\varphi_1)$</td>
</tr>
<tr>
<td>$\varphi_3$</td>
<td>$\varphi_3$</td>
<td>$\omega^3(\varphi_1)$</td>
</tr>
<tr>
<td>$\varphi_3$</td>
<td>$\varphi_3(1,0,0)$</td>
<td>$\omega^3(\varphi_1)$</td>
</tr>
<tr>
<td>$\varphi_3$</td>
<td>$\varphi_1(\omega,1)$</td>
<td>$\omega^3(\varphi_1)$</td>
</tr>
<tr>
<td>$\varphi_3$</td>
<td>$\varphi_1(\omega,1,0) + \varphi_1$</td>
<td>$\omega^3(\varphi_1)$</td>
</tr>
<tr>
<td>$\varphi_3$</td>
<td>$\varphi_{e0} + \varphi_1(\omega,1,0) + \varphi_1$</td>
<td>$\omega^3(\varphi_1)$</td>
</tr>
<tr>
<td>$\varphi_3(1,0,0)$</td>
<td>$\varphi_1$</td>
<td>$\omega^3(\varphi_1)$</td>
</tr>
<tr>
<td>$\varphi_3(1,0,0)$</td>
<td>$\varphi_3$</td>
<td>$\omega^3(\varphi_1)$</td>
</tr>
<tr>
<td>$\varphi_3(1,0,0)$</td>
<td>$\varphi_3(1,0,0)$</td>
<td>$\omega^3(\varphi_1)$</td>
</tr>
<tr>
<td>$\varphi_3(1,0,0)$</td>
<td>$\varphi_1(\omega,1)$</td>
<td>$\omega^3(\varphi_1)$</td>
</tr>
<tr>
<td>$\varphi_3(1,0,0)$</td>
<td>$\varphi_1(\omega,1,0) + \varphi_1$</td>
<td>$\omega^3(\varphi_1)$</td>
</tr>
<tr>
<td>$\varphi_3(1,0,0)$</td>
<td>$\varphi_{e0} + \varphi_1(\omega,1,0) + \varphi_1$</td>
<td>$\omega^3(\varphi_1)$</td>
</tr>
<tr>
<td>$\varphi_1(\omega,1)$</td>
<td>$\varphi_1$</td>
<td>$\omega^3(\varphi_1)$</td>
</tr>
<tr>
<td>$\varphi_1(\omega,1)$</td>
<td>$\varphi_3$</td>
<td>$\omega^3(\varphi_1)$</td>
</tr>
<tr>
<td>$\varphi_1(\omega,1)$</td>
<td>$\varphi_3(1,0,0)$</td>
<td>$\omega^3(\varphi_1)$</td>
</tr>
<tr>
<td>$\varphi_1(\omega,1)$</td>
<td>$\varphi_1(\omega,1)$</td>
<td>$\omega^3(\varphi_1)$</td>
</tr>
<tr>
<td>$\varphi_1(\omega,1)$</td>
<td>$\varphi_1(\omega,1,0) + \varphi_1$</td>
<td>$\omega^3(\varphi_1)$</td>
</tr>
<tr>
<td>$\varphi_1(\omega,1)$</td>
<td>$\varphi_{e0} + \varphi_1(\omega,1,0) + \varphi_1$</td>
<td>$\omega^3(\varphi_1)$</td>
</tr>
<tr>
<td>$\varphi_1(\omega,1,0) + \varphi_1$</td>
<td>$\varphi_1$</td>
<td>$\omega^3(\varphi_1)$</td>
</tr>
<tr>
<td>$\varphi_{e0} + \varphi_1(\omega,1,0) + \varphi_1$</td>
<td>$\varphi_3$</td>
<td>$\omega^3(\varphi_1)$</td>
</tr>
<tr>
<td>$\varphi_{e0} + \varphi_1(\omega,1,0) + \varphi_1$</td>
<td>$\varphi_3(1,0,0)$</td>
<td>$\omega^3(\varphi_1)$</td>
</tr>
<tr>
<td>$\varphi_{e0} + \varphi_1(\omega,1,0) + \varphi_1$</td>
<td>$\varphi_1(\omega,1)$</td>
<td>$\omega^3(\varphi_1)$</td>
</tr>
<tr>
<td>$\varphi_{e0} + \varphi_1(\omega,1,0) + \varphi_1$</td>
<td>$\varphi_1(\omega,1,0) + \varphi_1$</td>
<td>$\omega^3(\varphi_1)$</td>
</tr>
<tr>
<td>$\varphi_{e0} + \varphi_1(\omega,1,0) + \varphi_1$</td>
<td>$\varphi_{e0} + \varphi_1(\omega,1,0) + \varphi_1$</td>
<td>$\omega^3(\varphi_1)$</td>
</tr>
</tbody>
</table>

Table 23: IterFuncOrdinal exponential examples
8 Countable admissible ordinals

The first admissible ordinal is $\omega$. $\omega^{CK}$ (the Church-Kleene ordinal) is the second. This latter is the ordinal of the recursive ordinals. For simplicity it will be written as $\omega_1$ (see note 5). Gerald Sacks proved that the countable admissible ordinals are those defined like $\omega_1$, but using Turing Machines with oracles (see Note 9[17]. For example $\omega_2$ is the set of all ordinals whose structure can be enumerated by a TM with an oracle that defines the structure of all recursive ordinals.

One can think of these orderings in terms of well founded recursive functions on integers and subsets of integers. These subsets are neither recursive nor recursively enumerable but they are defined by properties of recursive processes. A first order well founded process is one that accepts an indefinite number of integer inputs and halts for every possible sequence of these inputs. Recursive ordinals are the well orderings definable by such a process. This definition can be iterated by defining a process of type $x + 1$ to be well founded for all sequences of processes of type $x$. (Type 0 is the integers.) This can be iterated up to any countable ordinal.

The admissible ordinals are a very sparse set of limit ordinals. In this document admissible level ordinals are all ordinals $\geq \omega_1$, the ordinal of the recursive ordinals. The ‘admissible index’ refers to the $\kappa$ index in $\omega_\kappa$.

8.1 Generalizing recursive ordinal notations

The ordinals definable in the countable admissible hierarchy can be defined as properties of recursive processes on non recursively enumerable domains. In turn these domains are defined as properties of recursive routines operating on lower level similarly defined domains. This continues down to the set of notations for recursive ordinals. This suggest an analogue to recursive ordinal notations for ordinal larger than $\omega_1$. This analog involves recursive function on finite symbols from a defined, but not recursively enumerable, domain. The idea is that a complete systems is the limit of an infinite sequences of extensions. The system can never be complete but it should be defined so that it can be expanded without limit.

Two reasons for exploring this hierarchy are its usefulness in expanding the hierarchy of recursive ordinals and its relevance to nondeterministic recursive processes. This approach can expand notation systems for recursive ordinals though a general form of ordinal collapsing. Modified names of admissible level ordinals are used to name larger recursive ordinals. We can always do this no matter how far we extend the hierarchy because the limit of the well orderings fully definable at any finite extension must be recursive. This follows because there is a recursive enumeration of all defined notations and a recursive process for deciding the relative size of any two defined notations.

There are formal system consistency problems or, equivalently, TM halting problems decidable by a specific recursive ordinal and not by smaller ones. However every halting problem is decidable by an ordinal $< \omega_1$, the ordinal of the recursive ordinals. Larger countable ordinals can decide questions that may be of relevance to finite beings in a divergent potentially infinite universe. For example, consider a universe that is finite at any time but potentially infinite and recursively deterministic. The question will a species have an

\[38\] A species, in contrast to an individuals direct descendants, can in theory have an infinite number of
infinite chain of descendant species is not in general decidable by a specific recursive ordinal because it requires quantification over the real to state. If we are concerned with questions that extend into an indefinite and possibly divergent future, the mathematics of countable admissible ordinals becomes essential.

8.2 Notations for Admissible level ordinals

Finite notation systems for recursive ordinals can always be expanded to represent larger ordinals. At the admissible level they can be expanded to partially fill gaps between pairs of definable ordinals. The following notations represent larger ordinals at all defined levels of the countable admissible hierarchy starting with the recursive ordinals. They also partially fill gaps between ordinal notations above the level of the recursive ordinals.

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

\[ \omega_{\kappa, \gamma}(\beta_1, \beta_2, ..., \beta_m) \]  \hspace{1cm} (8)

\[ \omega_{\kappa}[\eta] \]  \hspace{1cm} (9)

\[ [[\delta]]\omega_{\kappa}[\eta] \]  \hspace{1cm} (10)

\[ [[\delta]]\omega_{\kappa, \gamma}(\beta_1, \beta_2, ..., \beta_m) \]  \hspace{1cm} (11)

\[ [[\delta]]\omega_{\kappa}[\eta] \]  \hspace{1cm} (12)

The normal form notation for one term of an admissible level ordinal is given in expressions 8 to 12. These expressions add a parameter for the admissible index and a parameter in square brackets which signifies a smaller ordinal than the same term without an appended square bracketed parameter. There is a third option using double square brackets, [[]] as both a prefix and suffix to facilitate a form or collapsing described in Section 8.5.

The rest of the expression is the same as that for an IterFuncOrdinal shown in Expression 7 except \( \varphi \) is replaced with \( \omega \) to signify that this is an admissible level notation. The C++ class for a single term of an admissible level ordinal notation is AdmisNormalElement and the class for an admissible level ordinal notation is AdmisLevOrdinal.

The \( \gamma \) and \( \beta_i \) parameters are defined as they are in expression 7. The existence of any of these parameters defines a larger ordinal than the same notation without them. In contrast to every parameter defined so far, the \( \eta \) parameter drops down to a lower level ordinal to both fill gaps in the notation system and to partially serve the defining function that limitElement has for recursive ordinal notations. These are explained in the next section. For example \( \omega_1 \) is the ordinal of the recursive ordinals but \( \omega_1[\alpha] \) is a recursive ordinal necessarily smaller than \( \omega_1 \) but larger than any recursive ordinal previously defined. The \( \eta \) parameter is only defined when \( \gamma \) and \( \beta_i \) are zero and \( \kappa \) is a successor.

The \( \delta \) parameter, like the \( \eta \) parameter, defines a smaller ordinal than the same expression without this parameter. Any expression that begins with \( [[\delta]] \) will be \( < \omega_\delta \) no matter how large other parameters may be. Because \( \omega_{\kappa} \) with \( \kappa \) as a limit is defined to be \( \bigcup_{\alpha < \kappa} \omega_{\alpha} \) it is problematic to define what is meant by a \( \delta \) prefix that is a limit. Thus \( \delta \) must always be a direct descendant species. Thus this question requires quantification over the reals to state.
successor $\leq \kappa$. In addition if $\delta = \kappa$ and the single bracketed $[\eta]$ suffix occurs, the $\delta$ prefix has no affect and is deleted.

The relationship between the $[[\delta]]$ prefix and other parameters requires elaboration. Every parameter of an ordinal with a $\delta$ prefix has the global value of $[[\delta]]$ applied to it unless it has a smaller $[[\delta]]$ prefix. A larger internal value for $\delta$, as part of a parameter, is not allowed.

### 8.3 Typed parameters and limitOrd

Notations defined in previous sections give the structure of ordinals through the `limitElement` and `compare` member functions. For admissible level ordinals, `compare` is adequate, although it is now operating on an incomplete domain. However `limitElement` can no longer define how a limit ordinal is built up from smaller ordinals. Admissible level notations for limit ordinals cannot in general be defined as the limit of ordinals represented by a recursive sequence of notations for smaller ordinals.

`limitOrd`, and `isValidLimitOrdParam` are defined as an analog to the `limitElement` Ordinal member function.

$$\alpha = \bigcup_{n \in \omega} \alpha . \text{limitElement}(n)$$


and

$$\alpha = \bigcup \beta : \alpha . \text{isValidLimitOrdParam}(\beta) \alpha . \text{limitOrd}(\beta)$$

The system is designed so that the notations, $\beta$, that satisfy $\omega_1. \text{isValidLimitOrdParam}(\beta)$ can be expanded to include notations for any recursive ordinal. In general the notations, $\beta$, satisfying $\omega_\alpha. \text{isValidLimitOrdParam}(\beta)$ should be expandable to represent any ordinal $< \omega_\alpha$.

There are three functions used by `isValidLimitOrdParam`: `limitType`, `maxLimitType`, and `embedType`. Successor ordinal notations have `limitType` = 0 or `nullLimitType`. Notations for recursive limit ordinals represent the union of ordinals represented by a recursive sequence of notations for smaller ordinals. These have 1 (or `integerLimitType`) as their `limitType`. $\omega_1. \text{limitType} = 2$ or `recOrdLimitType`. $\omega_n. \text{limitType} = n + 1$. For infinite $\alpha$, $\omega_\alpha. \text{limitType} = \alpha$. $\alpha. \text{maxLimitType}$ is the maxima of $\beta. \text{limitType}$ for $\beta \leq \alpha$. For ordinals, $\alpha$ without a $[[\delta]]$ prefix: $\alpha. \text{isValidLimitOrdParam}(\beta) \equiv \alpha. \text{limitType} > \beta. \text{maxLimitType}$. Any ordinal with prefix $[[\delta]]$ must be $< \omega_\delta$. This puts upper bounds on both `limitType` and `maxLimitType`.

`embedType` returns a null pointer and is ignored unless $\delta = \kappa$ in $[[\delta]][\omega_\delta]$. It is only used to facilitate ordinal collapsing as discussed in Section 9.3. $\alpha. \text{isValidLimitOrdParam}(\beta)$ first tests if $\beta. \text{maxLimitType}() < \alpha. \text{limitType}()$. If this is true it return true. If they are equal and `embedType` is not null the boolean expression $\alpha > \beta$ is returned.

All of these new member functions are recursive in the domain of defined ordinal notations. Because this domain is incomplete at and beyond some recursive ordinal these functions need to be expanded as the notations are expanded. In C++ that can be accomplished

---

39 The equalities refer to the notations represented by the ordinal notations.

40 `limitOrd`, `limitType`, and `maxLimitType` are all member functions in the ordinal calculator. In addition `embedType`, if it is not empty, plus the other functions mentioned have their values displayed by the `limitExitCode` or alternatively the `lec` ordinal calculator member function.

41 `embedType` if non null is a pointer to an ordinal one greater than `limitType` but this value does not need to be tested in `.isValidLimitOrdParam`. 

46
nullLimitType=0, integerLimitType=1, recOrdLimitType=2

<table>
<thead>
<tr>
<th>$\alpha$</th>
<th>$\alpha$.limitType</th>
<th>$\alpha$.maxLimitType</th>
</tr>
</thead>
<tbody>
<tr>
<td>integer $\geq 0$</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>infinite recursive ordinal ($&lt;\omega_1$)</td>
<td>$\leq 1$</td>
<td>1</td>
</tr>
<tr>
<td>recursive limit ordinal</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>$\omega_1 \leq \alpha &lt; \omega_2$</td>
<td>$\leq 2$</td>
<td>2</td>
</tr>
<tr>
<td>$\omega_n \leq \alpha &lt; \omega_{n+1}$</td>
<td>$\leq n+1$</td>
<td>$n+1$</td>
</tr>
<tr>
<td>$\alpha = \omega_\omega$</td>
<td>1</td>
<td>$\omega_\omega$</td>
</tr>
<tr>
<td>$\omega_\omega+5 \leq \alpha &lt; \omega_\omega+6$</td>
<td>$\leq \omega+5$</td>
<td>$\omega+5$</td>
</tr>
<tr>
<td>$\alpha = \omega_\omega_1$</td>
<td>2</td>
<td>$\omega_\omega_1$</td>
</tr>
<tr>
<td>$\omega_\omega_1+1 \leq \alpha &lt; \omega_\omega_1+2$</td>
<td>$\leq \omega_1+1$</td>
<td>$\omega_1+1$</td>
</tr>
</tbody>
</table>

Table 24: limitType and maxLimitType examples

by making them virtual functions. The notation system should be expanded so the existing limitOrd function continues to work for parameters that meet the isValidLimitOrdParam constraint.

For example the limitOrd function applied to the notation for the Church-Kleene ordinal must be able to operate on an expanded notation system that could include a notation for any recursive ordinal. Because $\omega_1$.limitType = 2 and maxLimitType for any recursive ordinal is 1 or 0, there is a trivial way to do this. Define $\omega_1$.limitOrd to be the identity function that accepts any notation for a recursive ordinal as input and outputs its input. Then the union of the outputs for inputs satisfying this criteria is $\omega_1$. This is not the way limitOrd is defined. It is used to define new ordinals that help to fill the gaps between admissible levels, but this illustrates how limitOrd is able to partially provide the defining function that limitElement serves for recursive ordinal notations.

Table 24 shows the value of limitType and maxLimitType over ranges of ordinal values. The explicitly typed hierarchy in this document is a bit reminiscent of the explicitly typed hierarchy in Whitehead and Russell’s Principia Mathematica[19].

The idea of the $\delta$ parameter is to embed the notation system within itself in complex ways a bit like the Mandelbrot set[13] is embedded in itself. Table 25 gives the equations that define the the $\eta$ suffix with single and double square brackets. The top two line of this table define $[[\delta]]\omega_\kappa[\eta]$ and $[[\delta]]\omega_\kappa[[\eta]]$ for $\eta$ a limit in an obvious way. A single bracketed suffix (as in the first case) requires that $\delta \neq \kappa$. In all cases $\delta \leq \kappa$.

The rest of the table deals with $\eta$ a successor. For various values of $\delta, \kappa$ and ($[\eta]$ or $[[\eta]]$) the table defines $\alpha_0$ and $\alpha_i$ such that either

$$[[\delta]]\omega_\kappa[\eta] = \bigcup_{i \in \omega} \alpha_i$$

or

$$[[\delta]]\omega_\kappa[[\eta]] = \bigcup_{i \in \omega} \alpha_i.$$  

The Rf column of this table refers to lines in table 26 with examples.
For η a limit \([\delta]ω_κ[η] = \bigcup_{\mu \in η} ([\delta]ω_κ[\mu])\)
and \([\delta]ω_κ[η] = \bigcup_{\mu \in η} ([\delta]ω_κ[\mu])\)

\(\mathbf{Sp}(\alpha)\) says \(\alpha\) is a successor \(> 1\).

\(κ\) must be a successor with nonzero \(η\).

1p1 means \(\text{limPlus}_1\) which avoids a fixed point by adding 1
if psuedoCodeLevel > cantorCodeLevel (Section 6.3).

The \(Rf\) column is the line in Table 26 of an example.

For \(δ, κ\) and \(η\) (satisfying \(η\)) this table defines \(α_0\) and \(α_ι\)
such that \([\delta]ω_κ[η] = \bigcup_{i ∈ ω} α_ι\) or \([\delta]ω_κ[η] = \bigcup_{i ∈ ω} α_ι\).

<table>
<thead>
<tr>
<th>(δ)</th>
<th>(κ)</th>
<th>(η)</th>
<th>(α_0)</th>
<th>(Rf)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>(ω)</td>
<td>(\varphi_{α_ι,1p1})</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>(\mathbf{Sp}(κ))</td>
<td>1</td>
<td>(ω_{κ-1})</td>
<td>(ω_{κ-1,α_ι,1p1})</td>
</tr>
<tr>
<td>(δ = κ)</td>
<td>(κ)</td>
<td>(η)</td>
<td>Not allowed: ([η]) requires (δ &lt; κ)</td>
<td>11</td>
</tr>
<tr>
<td>(δ = κ)</td>
<td>(κ)</td>
<td>([η &gt; 1])</td>
<td>([δ]ω_κ[η] − 1])</td>
<td>(ω_{κ-1,α_ι})</td>
</tr>
<tr>
<td>(δ &lt; κ)</td>
<td>(κ)</td>
<td>([η &gt; 1])</td>
<td>([δ]ω_κ[η − 1])</td>
<td>(δ]ω_κ[α_ι])</td>
</tr>
<tr>
<td>(δ &lt; κ)</td>
<td>(κ)</td>
<td>([η &gt; 1])</td>
<td>([δ]ω_κ[η] − 1])</td>
<td>(δ]ω_κ[α_ι])</td>
</tr>
</tbody>
</table>

Table 25: Equations for \([\delta]\), \([η]\) and \([η]\)

<table>
<thead>
<tr>
<th>(Rf)</th>
<th>Ordinal</th>
<th>(\text{limitElements})</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>(ω_1)</td>
<td>(1)</td>
</tr>
<tr>
<td>2</td>
<td>([1]ω_1)</td>
<td>(\varphi_ω)</td>
</tr>
<tr>
<td>3</td>
<td>([2]ω_1)</td>
<td>(\varphi_{2}ω)</td>
</tr>
<tr>
<td>4</td>
<td>([3]ω_1)</td>
<td>(\varphi_{3}ω)</td>
</tr>
<tr>
<td>5</td>
<td>([4]ω_1)</td>
<td>(\varphi_{4}ω)</td>
</tr>
<tr>
<td>6</td>
<td>([5]ω_1)</td>
<td>(\varphi_{5}ω)</td>
</tr>
<tr>
<td>7</td>
<td>([6]ω_1)</td>
<td>(\varphi_{6}ω)</td>
</tr>
<tr>
<td>8</td>
<td>([7]ω_1)</td>
<td>(\varphi_{7}ω)</td>
</tr>
<tr>
<td>9</td>
<td>([8]ω_1)</td>
<td>(\varphi_{8}ω)</td>
</tr>
<tr>
<td>10</td>
<td>([9]ω_1)</td>
<td>(\varphi_{9}ω)</td>
</tr>
<tr>
<td>11</td>
<td>([10]ω_1)</td>
<td>(\varphi_{10}ω)</td>
</tr>
<tr>
<td>12</td>
<td>([11]ω_1)</td>
<td>(\varphi_{11}ω)</td>
</tr>
<tr>
<td>13</td>
<td>([12]ω_1)</td>
<td>(\varphi_{12}ω)</td>
</tr>
<tr>
<td>14</td>
<td>([13]ω_1)</td>
<td>(\varphi_{13}ω)</td>
</tr>
</tbody>
</table>

Table 26: \([\delta]\), \([η]\) and \([η]\) parameter examples in increasing order

48
8.4 limitType of admissible level notations

The $\kappa$ parameter in expressions 8 to 12 determines the limitType of an admissible level notation with no other nonzero parameters. If $\kappa$ is a limit then $\omega_\kappa$.limitType() = $\kappa$.limitType(). If $\kappa$ is a finite successor then $\omega_\kappa$.limitType() = $\kappa + 1$. If $\kappa > \omega$ and $\kappa$ is a successor then $\omega_\kappa$.limitType() = $\kappa$.

If an admissible level ordinal, $\alpha$, has parameters other than $\kappa$, then the one or two least significant nonzero parameters determine the limitType of $\alpha$. If the least significant parameter, $\alpha_1$, is a limit ordinal then $\alpha$.limitType = $\alpha_1$.limitType. $\alpha$.limitType = 1 or integerLimitType of any of the following conditions hold.

- $\gamma$ is a successor and the least significant parameter.
- There is at least one $\beta_i = 0$ and a more significant $\beta_i$ that is the least significant nonzero parameter.
- The least significant two parameters are successors.

If the above do not hold and the least significant parameter is a successor and the next least significant, $\alpha_n$, is a limit then $\alpha$.limitType = $\alpha_n$.limitType. There is a more complete description of these rules and their effects on limitElement and limitOrd in sections 9.2 and 9.6.

8.5 Ordinal collapsing

Collapsing expands recursive ordinal notations using larger ordinals (countable or uncountable) to name them[2, 15]. There is an element of collapsing with the $\eta$ parameter in square brackets defined in Section 8.2 in that it diagonalizes the earlier definitions of recursive ordinals by referencing a higher level notation. Before describing collapsing with the $\delta$ parameter, we give a brief overview of an existing approach.

One can define a collapsing function, $\Psi(\alpha)$ on ordinals to countable ordinals[43]. $\Psi(\alpha)$ is defined using a function $C(\alpha)$, from ordinals to sets of ordinals. $C(\alpha)$ is defined inductively on the integers for each ordinal $\alpha$ using $\Psi(\beta)$ for $\beta < \alpha$.

- $C(\alpha)_0 = \{0, 1, \omega , \Omega\}$ ($\Omega$, is the ordinal of the countable ordinals.)
- $C(\alpha)_{n+1} = C(\alpha)_n \cup \{\beta_1 + \beta_2 , \beta_1 \beta_2 , \beta_1^{\beta_2} : \beta_1 , \beta_2 \in C(\alpha)_n \} \cup \{\Psi(\beta) : \beta \in C(\alpha)_n \}$
- $C(\alpha) = \bigcup_{n \in \omega} C(\alpha)_n$.

Ordinal collapsing is also known as projection. I prefer the term collapsing, because I think the proofs have more to do with syntactical constructions of mathematical language than they do with an abstract projection in a domain the symbols refer to. Specifically I think cardinal numbers, on which projection is most often based, have only a relative meaning. See Section 12 for more about this philosophical approach.

This description is largely based on the Wikipedia article on “Ordinal collapsing function” as last modified on April 14, 2009 at 15:48. The notation is internally consistent in this document and differs slightly from the Wikipedia notation.
• \( \Psi(\alpha) \) is defined as the least ordinal not in \( C(\alpha) \).

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

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

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

Collapsing, as defined here, connects basic ordinal arithmetic (addition, multiplication and exponentiation) to higher level ordinal functions. Ordinal arithmetic on \( \Omega \) gets past fixed points in the definition of \( \Psi \) until we reach an ordinal that is not in \( C(\alpha) \) either by definition or by basic ordinal arithmetic on ordinals in \( C(\alpha) \). This is the ordinal \( \varepsilon_{\Omega+1}^{44} \). \( \Psi(\varepsilon_{\Omega+1}^{44}) = \Psi(\Omega) \cup \Psi(\Omega^\Omega) \cup \Psi(\Omega^\Omega^\Omega) \cup \Psi(\Omega^\Omega^\Omega^\Omega) \cup ... \) This is the Bachmann-Howard ordinal[11]. It is the largest ordinal in the range of \( \Psi \) as defined above. \( \Psi(\alpha) \) is a constant for \( \alpha \geq \varepsilon_{\Omega+1} \) because there is no way to incorporate \( \Psi(\varepsilon_{\Omega+1}) \) into \( C(\alpha) \).

\( ^{44} \text{Note } \varepsilon_{\Omega} = \Omega. \)
<table>
<thead>
<tr>
<th>#</th>
<th>( \Psi )</th>
<th>( \varphi )</th>
<th>( \omega )</th>
<th>Bound</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>( \Psi(\alpha) )</td>
<td>( \varepsilon_\alpha )</td>
<td></td>
<td>( \alpha &lt; \varphi(2,0) )</td>
</tr>
<tr>
<td>2</td>
<td>( \Psi(\Omega^{13}) )</td>
<td>( \varphi(2,12) )</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>( \Psi(\Omega(1+\alpha)) )</td>
<td>( \varphi(2,\alpha) )</td>
<td></td>
<td>( \alpha &lt; \varphi(3,0) )</td>
</tr>
<tr>
<td>4</td>
<td>( \Psi(\Omega^26) )</td>
<td>( \varphi(3,5) )</td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>( \Psi(\Omega^2(1+\alpha)) )</td>
<td>( \varphi(3,\alpha) )</td>
<td></td>
<td>( \alpha &lt; \varphi(4,0) )</td>
</tr>
<tr>
<td>6</td>
<td>( \Psi(\Omega^36) )</td>
<td>( \varphi(12,5) )</td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>( \Psi(\Omega^2(1+\alpha)) )</td>
<td>( \varphi(1+\beta,\alpha) )</td>
<td>( \alpha &lt; \varphi(1+\beta,0) \wedge \beta &lt; \varphi(1,0,0) )</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>( \Psi(\Omega^4) )</td>
<td>( \varphi(1,0,0) = \Gamma_0 )</td>
<td></td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>( \Psi(\Omega^{12}) )</td>
<td>( \varphi(2,0,0) )</td>
<td></td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>( \Psi(\Omega^{12+3}6) )</td>
<td>( \varphi(2,3,5) )</td>
<td></td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>( \Psi(\Omega^{14}) )</td>
<td>( \varphi(1,0,0,0) )</td>
<td></td>
<td></td>
</tr>
<tr>
<td>12</td>
<td>( \Psi(\Omega^{15}) )</td>
<td>( \varphi(1,0,2,...,0_{n+2}) )</td>
<td></td>
<td></td>
</tr>
<tr>
<td>13</td>
<td>( \Psi(\Omega^{3^{n+2}}\alpha) )</td>
<td>( \varphi(\alpha_1,0_2,...,0_{n+2}) )</td>
<td>( \alpha_1 &lt; \varphi(1_1,0_2,...,0_{n+3}) )</td>
<td></td>
</tr>
<tr>
<td>14</td>
<td>( \Psi(\Omega^{15}) )</td>
<td>( \varphi_1 )</td>
<td></td>
<td></td>
</tr>
<tr>
<td>15</td>
<td>( \Psi(\Omega^{15+\alpha}) )</td>
<td>( \varphi_1(5) )</td>
<td></td>
<td></td>
</tr>
<tr>
<td>16</td>
<td>( \Psi(\Omega^{15}(1+\alpha)) )</td>
<td>( \varphi_1(\alpha) )</td>
<td></td>
<td>( \alpha &lt; \varphi_1(1,0) )</td>
</tr>
<tr>
<td>17</td>
<td>( \Psi(\Omega^{15}(1+\alpha+2)) )</td>
<td>( \varphi_1(4,2) )</td>
<td></td>
<td></td>
</tr>
<tr>
<td>18</td>
<td>( \Psi(\Omega^{15}) )</td>
<td>( \varphi_2 )</td>
<td></td>
<td></td>
</tr>
<tr>
<td>19</td>
<td>( \Psi(\Omega^{15}) )</td>
<td>( \varphi_\omega )</td>
<td></td>
<td>( \alpha &lt; \omega [1] )</td>
</tr>
<tr>
<td>20</td>
<td>( \Psi(\Omega^{15}) )</td>
<td>( \varphi_\omega )</td>
<td></td>
<td></td>
</tr>
<tr>
<td>21</td>
<td>( \Psi(\Omega^{15}) )</td>
<td>( \varphi_\omega )</td>
<td></td>
<td>( \alpha &lt; \omega_1 [1] )</td>
</tr>
<tr>
<td>22</td>
<td>( \Psi(\Omega^{15}) )</td>
<td>( \varphi_\omega )</td>
<td></td>
<td>( \alpha &lt; \omega_1 [1] )</td>
</tr>
<tr>
<td>23</td>
<td>( \Psi(\Omega^{15}) )</td>
<td>( \varphi_\omega )</td>
<td></td>
<td>( \alpha &lt; \omega_1 [1] )</td>
</tr>
<tr>
<td>24</td>
<td>( \Psi(\varepsilon_{\Omega+1}) )</td>
<td>( \varphi_\omega )</td>
<td></td>
<td>( \alpha &lt; \omega_1 [1] )</td>
</tr>
</tbody>
</table>

Table 27: \( \Psi \) collapsing function with bounds
<table>
<thead>
<tr>
<th>Notation</th>
<th>$\Psi(\Omega)$</th>
<th>$\Psi(\Omega^2)$</th>
<th>$\Psi(\Omega^2 + 1)$</th>
<th>$\Psi(\Omega^2 + 2)$</th>
<th>$\Psi(\Omega^2 + 3)$</th>
<th>$\Psi(\Omega^2 + 4)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\Phi_0$</td>
<td>$\Phi_1$</td>
<td>$\Phi_2$</td>
<td>$\Phi_3$</td>
<td>$\Phi_4$</td>
<td>$\Phi_5$</td>
<td>$\Phi_6$</td>
</tr>
<tr>
<td>$\Phi_1$</td>
<td>$\Phi_2$</td>
<td>$\Phi_3$</td>
<td>$\Phi_4$</td>
<td>$\Phi_5$</td>
<td>$\Phi_6$</td>
<td>$\Phi_7$</td>
</tr>
<tr>
<td>$\Phi_2$</td>
<td>$\Phi_3$</td>
<td>$\Phi_4$</td>
<td>$\Phi_5$</td>
<td>$\Phi_6$</td>
<td>$\Phi_7$</td>
<td>$\Phi_8$</td>
</tr>
</tbody>
</table>

Table 28: $\Psi$ collapsing function at critical limits
Table 27 connects the notations defined by the Ψ function to those defined with expressions 4, 6, 7 and 8 to 12. Lines up to 12 follow from the above description. This line uses parameter subscripts to indicate the number of parameters. Otherwise it is straightforward. In line 14 $\varphi_1$ is the first fixed point not reachable from Veblen functions with a finite number of parameters. $\varphi_1 = \bigcup_{n \in \omega} \varphi(1, 0_2, 0_3, \ldots, 0_n)$. Lines 18, 19 and 20 illustrate how each increment of $\alpha$ by one in $\varphi_\alpha$ adds a factor of $\omega$ to the the highest order exponent in Ψ notation. This maps to the definition of $\varphi_\alpha$ in Section 5.5.

Table 28 provides additional detail to relate the Ψ notation to the notation in this document. It covers the range of the Ψ function at critical limits. For each entry the table gives the ordinal notation and first 4 elements in a limit sequence that converges to this ordinal. This same information is repeated in paired lines. The first is in Ψ notation and the second in the notation defined in this document. It illustrates the conditions in which the Ψ notation requires another level of exponentiation. This happens with limiting sequences that involve ever higher levels of exponentiation of the largest ordinals defined at that level in the Ψ notation. This occurs in lines 4, 9, 11 and 12 of the table corresponding to $\Omega$

8.6 Displaying Ordinals in Ψ format

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

8.7 Admissible level ordinal collapsing

For any two ordinals $\omega_\alpha$ and $\omega_\alpha+1$ there is an unclosable gap into which one can embed the ordering structure of any finite recursive notation system. As mentioned before finite formulations of a fragment of the countable admissible level hierarchy are a bit like the Mandelbrot set where the entire structure is embedded again and again at many points in an infinite tree.

The idea is to “freeze” the structure at some point in its development and then embed this frozen image in an unfrozen version of itself. It is a bit like taking recursion to a higher level of abstraction. The notation is frozen by not allowing any values beyond the frozen level as $\eta$ parameters. It puts no constraints on the other parameters. In a sense it brings the entire hierarchy of notations at this stage of development into representations for ordinals less than the level at which the notation system is frozen.

The relevant expressions are $10(\lbrack \delta \rbrack \omega_\kappa[n])$, $11(\lbrack \delta \rbrack \omega_{\kappa, \gamma}(\beta_1, \beta_2, \ldots, \beta_m))$ and $12(\lbrack \delta \rbrack \omega_\kappa[n])$. They define a value $< \omega_\delta$ regardless of the size of $\kappa$ and other parameters (excluding $\eta$ which is constrained by $\delta$). It is required that $\kappa \geq \delta$. The idea is to use notations up

\footnote{Note that any notation from expressions 10, 11 and 12 with prefix \([1]\) must define a recursive ordinal.}
through any $\omega_\kappa$ definable in the frozen system to define ordinal notations $< \omega_6$ in the unfrozen system.

This starts by diagonalizing what can be defined with $\omega_\kappa[\eta]$. See Table 29 for the definition of $[[1]]\omega_1$. The complete definition of $\delta$ (and the other parameters in expressions 10, 11 and 12) are in sections 9.2 and 9.6 that document the limitElement and limitOrd member functions.
<table>
<thead>
<tr>
<th>$\alpha$</th>
<th>$n$</th>
<th>$1$</th>
<th>$2$</th>
<th>$3$</th>
<th>$4$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$[[1]\omega_1]$</td>
<td>$[[1]\omega_1[1]]$</td>
<td>$[[1]\omega_1[2]]$</td>
<td>$[[1]\omega_1[3]]$</td>
<td>$[[1]\omega_1[4]]$</td>
<td></td>
</tr>
<tr>
<td>$[[1]\omega_1[1]]$</td>
<td>$\omega$</td>
<td>$\omega$</td>
<td>$\omega$</td>
<td>$\omega$</td>
<td></td>
</tr>
<tr>
<td>$[[1]\omega_1[2]]$</td>
<td>$[[1]\omega_1[1]]$</td>
<td>$[[1]\omega_1[1]]$</td>
<td>$[[1]\omega_1[1]]$</td>
<td>$[[1]\omega_1[1]]$</td>
<td></td>
</tr>
<tr>
<td>$[[1]\omega_1[3]]$</td>
<td>$[[1]\omega_1[2]]$</td>
<td>$[[1]\omega_1[2]]$</td>
<td>$[[1]\omega_1[2]]$</td>
<td>$[[1]\omega_1[2]]$</td>
<td></td>
</tr>
<tr>
<td>$[[1]\omega_1[4]]$</td>
<td>$[[1]\omega_1[3]]$</td>
<td>$[[1]\omega_1[3]]$</td>
<td>$[[1]\omega_1[3]]$</td>
<td>$[[1]\omega_1[3]]$</td>
<td></td>
</tr>
</tbody>
</table>


Table 29: Structure of $[[1]\omega_1]$
9 AdmisLevOrdinal class

The AdmisLevOrdinal class constructs notations for countable admissible ordinals beginning with the Church-Kleene or second admissible ordinal\(^{46}\). This is the ordinal of the recursive ordinals. The AdmisLevOrdinals are different from ordinals constructed with its base classes. For every recursive ordinal there is a finite notation system that can enumerate a notation for that ordinal and every ordinal less than it. This is not true for the ordinal of the recursive ordinals or larger ordinals.

Four new member functions: `limitOrd`, `isValidLimitOrdParam`, `limitType`, `embedType` and `maxLimitType` support this level of the ordinal hierarchy. These functions are outlined in Section 8 and described in detail in sections 9.3 to 9.5.

C++ class AdmisLevOrdinal uses an expanded version of the normal form in expression 7 given in expressions 8 to 12. The class for a single term of an AdmisLevOrdinal is AdmisNormalElement. It is derived from IterFuncNormalElement and its base classes.

All parameters (except \(\kappa\)) can be zero. Omitted parameters are set to 0. To omit \(\kappa\) use the notation for an IterFuncOrdinal in expression 7. \(\kappa\) is the admissible index, i.e. \((\kappa \in \omega_\kappa)\). \(\omega_1\) is the ordinal of the recursive ordinals or the smallest ordinal that is not recursive. The expanded notation at this level is specified in Section 8.2. Parameters \(\gamma\) and \(\beta_i\) have a definition similar to that in Table 10 and are explained in section 9.2 on AdmisNormalElement::limitElement and section 9.6 on AdmisNormalElement::limitOrd as are the new parameters in this class: \(\kappa, \eta\) and \(\delta\).

The AdmisLevOrdinal class should not be used directly to create ordinal notations. Instead use function `admisLevelFunctional` which checks for fixed points and creates unique notations for each ordinal\(^{47}\). This function takes up to five arguments. The first two are required and the rest are optional. The first gives the admissible ordinal index, the \(\kappa\) in \(\omega_\kappa\). The second gives the level of iteration or the value of \(\gamma\) in expressions 8 and 11. The third gives a NULL terminated array of pointers to Ordinals which are the \(\beta_i\) in expressions 8 and 11. This parameter is optional and can be created with function `createParameters` described in Section 6. The fourth parameter is an ordinal reference that defaults to 0. It is the value of \(\eta\) in expressions 9 and 10. The fifth parameter is an Ordinal reference which defaults to zero. It is the value of \(\delta\) in expressions 10, 11 and 12. Some examples are shown in Table 30. An alternative way of defining these ordinals in C++ format is with the `cppList` command in the ordinal calculator. Some examples of this output are shown in Figure 1.

9.1 compare member function

AdmisNormalElement::compare is called from a notation for one term in the form of expressions 8 to 12. It compares the CantorNormalElement parameter `trm`, against the AdmisNormalElement instance `compare` is called from. As with FiniteFuncOrdinals and

---

\(^{46}\)The first admissible ordinal is the ordinal of the integers, \(\omega\).

\(^{47}\)In the interactive ordinal calculator one can use notations like `[[delta]] omega,{ kappa, lambda}` `(b1,b2,...,bn)`. For more examples see sections B.11.7, B.11.8 and B.11.9.
const Ordinal& a = expFunctional(Ordinal::one) ;

const Ordinal& b = iterativeFunctional(
    Ordinal::one,
    Ordinal::one,
    Ordinal::zero) ;

const Ordinal& c = admisLevelFunctional(
    Ordinal::one,
    Ordinal::zero,
    createParameters(
        &(Ordinal::one),
        &(Ordinal::one),
        &(Ordinal::zero))
) ;

const Ordinal& d = admisLevelFunctional(
    Ordinal::one,
    Ordinal::zero,
    NULL
) ;

const Ordinal& e = admisLevelFunctional(
    Ordinal::one,
    Ordinal::zero,
    NULL,
    psi( Ordinal::one, Ordinal::zero)
) ;

Figure 1: Defining AdmisLevOrdinals with cppList
C++ code examples | Ordinal
---|---
af(zero, zero, cp(&one)) | $\omega_1(1,0)$
af(zero, one, cp(&one, &zero)) | $\omega_1(1,1,0)$
af(one, zero, cp(&one, &one, &zero)) | $\omega_1$
af(one, zero) | $\omega_1[\varepsilon_0]$
af(one, zero, NULL, eps0) | $\omega_1[5]$ 
af(one, NULL, Ordinal::five) | $\omega_1$ 
af(one, omega1CK, cp(&one)) | $\omega_1[1]$ 
af(one, one, cp(&one, &omega1CK)) | $\omega_1[1](1,\omega_1)$ 
af(one, Ordinal::two, cp(&one, &one, &omega, &zero)) | $\omega_1[2](1,\omega,0)$
af(omega, omega, cp(&one, &omega1CK, &zero, &zero)) | $\omega_1[\omega_1,\omega](1,\omega_1,0,0)$ 
af(omega1CK, omega) | $\omega_1[\omega_1,\omega](\varphi_\omega)$
af(one, omega, cp(&iterativeFunctional(omega))) | $\omega_1[\omega_1,\omega](\varphi_\omega)$
af(one, Ordinal::two, NULL, zero) | $\omega_1$ 
af(Ordinal::two, zero, NULL, zero, Ordinal::two) | $\omega_2$
af(Ordinal::two, one, NULL, zero, Ordinal::three) | $\omega_3$
af(omega, zero, NULL, omega) | $\omega_2$ 
af(Ordinal::two, omega1CK, cp(&one), zero, Ordinal::two) | $\omega_2[1](1)$ 
af(omega, omega, cp(&one, &omega1CK, &zero), zero, one) | $\omega_2[1](1,\omega_1,0)$ 

Table 30: admisLevelFunctional C++ code examples

IterFuncOrdinals, the work of comparing Ordinals with multiple terms is left to the Ordinal and OrdinalImpl base class functions.

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

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

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

58
<table>
<thead>
<tr>
<th>Symbol</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>cfp</td>
<td><code>compareFiniteParams</code> compares its $\beta_i$ to those in its argument</td>
</tr>
<tr>
<td>cmp</td>
<td><code>obj.compare(arg)</code> returns -1, 0 or 1 if <code>obj &lt;, = or &gt;</code> then <code>arg</code></td>
</tr>
<tr>
<td>compareCom</td>
<td>does comparisons used by higher level routines (see Table 33)</td>
</tr>
<tr>
<td>$\delta_{ef}$</td>
<td>effective (context dependent) value of $\delta$</td>
</tr>
<tr>
<td>diff</td>
<td>temporary value with compare result</td>
</tr>
<tr>
<td>dd</td>
<td><code>drillDown</code> $\eta$ in $[\eta]$ or $[[\eta]]$ suffix</td>
</tr>
<tr>
<td>effCK</td>
<td><code>effectiveIndexCK</code> returns $\kappa$ adjusted for $\delta$, $\eta$ and context.</td>
</tr>
<tr>
<td>functionLevel</td>
<td>$\gamma$ parameter in expressions 8 and 11</td>
</tr>
<tr>
<td>ignoreFactor</td>
<td>parameter to ignore factor in comparison</td>
</tr>
<tr>
<td>ignf</td>
<td><code>ignoreFactor</code> ignore factor and all but the first term</td>
</tr>
<tr>
<td>isDdE</td>
<td><code>isDrillDownEmbed</code> $\equiv$ has $[[\eta]]$ suffix</td>
</tr>
<tr>
<td>isZ</td>
<td><code>isZero</code> is value zero</td>
</tr>
<tr>
<td>maxParameter</td>
<td>largest parameter of class instance or <code>trm</code></td>
</tr>
<tr>
<td>maxParamFirstTerm</td>
<td>first term of largest parameter of <code>trm</code></td>
</tr>
<tr>
<td>parameterCompare</td>
<td>compare itself and its argument’s largest parameter and vice versa</td>
</tr>
<tr>
<td>this</td>
<td>pointer to <code>class</code> instance called from</td>
</tr>
<tr>
<td>trm</td>
<td>parameter comparing against</td>
</tr>
<tr>
<td>X</td>
<td>exit code (see Note 36 on page 38)</td>
</tr>
</tbody>
</table>

Table 31: Symbols used in `compare` tables 32 and 33

The function `compare` has four arguments.

- `const OrdinalImpl& embdIx` — the context for the base `class` object from which this `compare` originated.

- `const OrdinalImpl& termEmbdIx` — the context of the `CantorNormalElement` parameter of `compare`.

- `const CantorNormalElement& trm` — the term to be compared.

- `bool ignoreFactor` — an optional parameter that defaults to `false` and indicates that an integer `factor` is to be ignored in the comparison.

The original version of `compare`, without the context arguments, calls the routine with context parameters.

As with `compare` for `FiniteFuncOrdinals` and `IterFuncOrdinals` this function depends on the `getMaxParameter()` that returns the maximum value of all the parameters (except $\eta$ and $\delta$) in expressions 9, 11 and 12.

The logic of `AdmisNormalElement::compare` with the above four parameters is summarized in tables 32 and 33. The latter is for `AdmisNormalElement::compareCom` which contains comparisons suitable for use in higher level routines. It is called by `compare` with identical parameters for suitable tests. Definitions used in these two tables are contained in Table 31.
See Table 31 for symbol definitions.

Comparisons use δ context for both operands.

Value is $\text{ord.cmp}(\text{trm})$: -1, 0 or 1 if ord $<,=,>$ trm

<table>
<thead>
<tr>
<th>X</th>
<th>Condition</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>A1</td>
<td>if (trm.codeLevel &gt; admisCodeLevel) diff=-trm.cmp(*this)</td>
<td>diff</td>
</tr>
<tr>
<td>A2</td>
<td>(trm.codeLevel &lt; admisCodeLevel) applies to A2, A3 and A4</td>
<td>1</td>
</tr>
<tr>
<td>A3</td>
<td>(maxParamFirstTerm $\geq$ *this)</td>
<td>-1</td>
</tr>
<tr>
<td>A4</td>
<td>if neither the A2 or A3 condition is true</td>
<td>1</td>
</tr>
<tr>
<td>A5</td>
<td>(diff = parameterCompare(trm)) $\neq$ 0</td>
<td>diff</td>
</tr>
<tr>
<td>C1</td>
<td>if none of the above apply diff = compareCom(trm)</td>
<td>diff</td>
</tr>
</tbody>
</table>

Table 32: AdmisNormalElement::compare summary

See Table 31 for symbol definitions.

Comparisons use δ context for both operands.

Value is $\text{ord.cmp}(\text{trm})$: -1, 0 or 1 if ord $<,=,>$ trm

<table>
<thead>
<tr>
<th>X</th>
<th>Condition</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>B1</td>
<td>$\delta_{\text{ef}} \neq 0 \land \text{trm}.\delta_{\text{ef}} \neq 0 \land ((\text{diff} = \delta_{\text{ef}}.\text{cmp}(\text{trm}.\delta_{\text{ef}}) \neq 0)$</td>
<td>diff</td>
</tr>
<tr>
<td>B2</td>
<td>(diff=effCk.cmp(trm.effCK)) $\neq$ 0</td>
<td>diff</td>
</tr>
<tr>
<td>B3</td>
<td>(diff=indexCK.cmp(trm.indexCK)) $\neq$ 0</td>
<td>diff</td>
</tr>
<tr>
<td>B4</td>
<td>(dd$\neq 0) \land (\text{trm}.dd \neq 0) \land (\text{diff}=(\text{isDdE-trm.isDdE})\neq 0)</td>
<td>diff</td>
</tr>
<tr>
<td>B5</td>
<td>(dd$\neq 0) \land (\text{trm}.dd \neq 0) \land (\text{diff}=dd.cmp(\text{trm}.dd)\neq 0)</td>
<td>diff</td>
</tr>
<tr>
<td>B6</td>
<td>(diff=((dd$\neq 0) - (\text{trm}.dd \neq 0))$\neq 0)</td>
<td>diff</td>
</tr>
<tr>
<td>B7</td>
<td>(diff = $\gamma$.compare(trm.$\gamma$) $\neq$ 0</td>
<td>diff</td>
</tr>
<tr>
<td>B8</td>
<td>((diff = cfp(trm)) $\neq$ 0) $\lor$ ignf</td>
<td>diff</td>
</tr>
<tr>
<td>B9</td>
<td>(diff = factor - trm.factor) $\neq$ 0</td>
<td>diff</td>
</tr>
</tbody>
</table>

Table 33: AdmisNormalElement::compareCom summary
9.2 limitElement member function

AdmisNormalElement::limitElement overrides IterFuncNormalElement::limitElement (see Section 6.2). It operates on a single term of the normal form expansion and does the bulk of the work. It takes a single integer parameter. Increasing values for the argument yield larger ordinal notations as output. In contrast to base class versions of this routine, in this version the union of the ordinals represented by the outputs for all integer inputs are not necessarily equal to the ordinal represented by the AdmisNormalElement class instance limitElement is called from. They are equal if the limitType of this instance of AdmisNormalElement is integerLimitType (see tables 24 and 34).

As usual Ordinal::limitElement does the work of operating on all but the last term of an Ordinal by copying all but the last term of the result unchanged from the input Ordinal. The last term is generated based on the last term of the Ordinal instance limitElement is called from.

AdmisNormalElement::limitElement calls drillDownLimitElement when the $\eta$ or $[[\eta]]$ suffix is not zero. These two routines each has a version, limitElementCom and drillDownLimitElementCom, invoked by the base routine and suitable for routines at higher codeLevels to use. The common routines (ending with Com) always create a result using createVirtualOrd or createVirtualOrdImpl. These virtual functions supply any unspecified parameters whenever the com routine is called from objects at higher codeLevels.

The four ...limitElement... routines are outlined in tables 36 to 39. The tables give the LimitTypeInfo enum assigned to each limit when a new Ordinal is created. The code that computes a limitElement often starts with a case statement on instances of this enum. Table 34 gives the definition of these enums. The type of limit is determined by the one or two least significant nonzero parameters and these conditions are described in this table. The symbols used in all four tables for the variations of ...limitElement... are defined in Table 35.

Corresponding to these four routines are tables 40 to 43 of examples with exit codes that match those in the corresponding descriptive tables. The matching exit codes are in columns X or UpX depending on whether the exit code is for the routine that directly generated the result or one that calls a lower level routine to do so. The exit codes are in the same alphabetical order in both the descriptive and example tables. Table 44 gives an example for every value of LimitTypeInfo used by limitElement. Some additional examples are shown in Tables 45 to 47.

One complication is addressed by routine leUse. This is required when $\kappa$ is a limit and the least significant parameter. This routine selects an element from a sequence whose union is $\kappa$ while insuring that this does not lead to a value less than $\delta$. This selection must be chosen so that increasing inputs produce increasing outputs and the output is always less than the input. The same algorithm is used for limitOrd. See Note 48 on page 74 for a description of the algorithm. leUse is a virtual function so that routines that are using it (such as limitElementCom or limitOrdCom) may be callable from higher levels.
\[ \beta_i \text{ defined in: } 7, 8, 11 \]
\[ \gamma \text{ defined in: } 7, 8, 11 \]
\[ \eta \text{ defined in: } 9, 10, 12 \]
\[ \kappa \text{ defined in: } 8, 9, 10, 11, 12. \]
\[ \delta \text{ defined in: } 10, 11, 12. \]

<table>
<thead>
<tr>
<th>LimitTypeInfo</th>
<th>Least significant nonzero parameters</th>
<th>limitType</th>
</tr>
</thead>
<tbody>
<tr>
<td>unknownLimit</td>
<td>used internally</td>
<td></td>
</tr>
<tr>
<td>zeroLimit</td>
<td>the ordinal zero</td>
<td>nullLimitType</td>
</tr>
<tr>
<td>finiteLimit</td>
<td>a successor ordinals</td>
<td>nullLimitType</td>
</tr>
<tr>
<td>paramSucc</td>
<td>in ( \omega^x x ) is a successor</td>
<td>integerLimitType</td>
</tr>
</tbody>
</table>

The following is at code level `cantorCodeLevel` (Section 4) and above.

The following is at code level `finiteFuncCodeLevel` (Section 6) and above.

The following is at code level `iterFuncCodeLevel` (Section 7) and above.

The following is at code level `admisCodeLevel` (Section 9) and above.

| drillDownLimit | \[ \eta \] or \[ [\eta] \] suffix is a limit. | \( \eta\).limitType |
| drillDownSucc  | \( \eta \) is a successor \( \beta_i \). | integerLimitType |
| drillDownSuccCKOne | \( \kappa == 1 \) \& \( (\eta > 1) \lor \eta \) is a successor. | integerLimitType |
| drillDownSuccEmbed | \( \eta \) is a successor \( \beta_i \). | integerLimitType |
| drillDownOne | \( (\eta \equiv 1) \land (\kappa > 1) \). | integerLimitType |
| drillDownOneEmbed | \( \eta \equiv 1. \) | integerLimitType |
| drillDownCKOne | \( \eta \equiv \kappa \equiv 1 \). | integerLimitType |
| indexCKlimit | \( \kappa \) is a limit and \( \delta \equiv 0 \). | \( \kappa\).limitType |
| indexCKsuccParam | \( \kappa \) and a single \( \beta \) are successors \( \land \kappa \neq \delta \) | integerLimitType |
| indexCKsuccParamEq | \( \kappa \) and a single \( \beta \) are successors \( \land \kappa == \delta \) | integerLimitType |
| indexCKsuccEmbed | \( \kappa \) is a successor and \( \delta > 0 \). | indexCKtoLimitType |
| indexCKsuccUn | \( \kappa \) is a successor and \( \delta == 0 \). | indexCKtoLimitType |
| indexCKlimitParamUn | \( \kappa \) is a limit and \( \beta_i \) is a successor | \( \kappa\).limitType |
| indexCKlimitEmbed | \( \kappa \) is limit and \( \delta > 0 \). | \( \kappa\).limitType |

Table 34: LimitTypeInfo descriptions through admisCodeLevel
<table>
<thead>
<tr>
<th>Symbol</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\delta_{ck}$</td>
<td>if $\kappa = \delta$ use $\delta - 1$</td>
</tr>
<tr>
<td>dRepl</td>
<td>copy ordinal called from replacing $\eta$ parameter</td>
</tr>
<tr>
<td>IfNe</td>
<td>IterativeFunctionNormalElement see Section 7</td>
</tr>
<tr>
<td>isDdEmb</td>
<td>isDrillDownEmb (true iff $[[\eta]]$ suffix is nonzero)</td>
</tr>
<tr>
<td>isLm</td>
<td>isLimit</td>
</tr>
<tr>
<td>isSc</td>
<td>isSuccessor</td>
</tr>
<tr>
<td>indexCKtoLimitType</td>
<td>computes limitType $\kappa$ by adding 1 to $\kappa$ if it is finite</td>
</tr>
<tr>
<td>$\alpha.leUse(n)$</td>
<td>Compute a safe value for $\alpha le(n)$ and $lo(n)$ see Section 9.2</td>
</tr>
<tr>
<td>lSg</td>
<td>value of least significant nonzero $\beta_i$</td>
</tr>
<tr>
<td>lsx</td>
<td>index of least significant nonzero $\beta_i$</td>
</tr>
<tr>
<td>lo</td>
<td>limitOrd (Table 48)</td>
</tr>
<tr>
<td>lp1</td>
<td>limPlus_1 avoid fixed points by adding 1 if $psuedoCodeLevel &gt; CantorCodeLevel$ (Section 6.3)</td>
</tr>
<tr>
<td>n1Sg</td>
<td>value of next to least significant nonzero $\beta_i$</td>
</tr>
<tr>
<td>nlsx</td>
<td>index of next to least significant nonzero $\beta_i$</td>
</tr>
<tr>
<td>rep1</td>
<td>replace1, rep1(i, val) replaces $\beta_i$ with val in ord see Table 9</td>
</tr>
<tr>
<td>rep2</td>
<td>replace2, a two parameter version of rep1 see Table 9</td>
</tr>
<tr>
<td>Rtn x</td>
<td>x is return value</td>
</tr>
<tr>
<td>sz</td>
<td>number of $\beta_i$ in expressions 8 and 11</td>
</tr>
<tr>
<td>this</td>
<td>pointer to class instance called from</td>
</tr>
<tr>
<td>X</td>
<td>exit code (Note 36 on page 38)</td>
</tr>
</tbody>
</table>

Table 35: Symbols used in limitElement Tables 36 to 39 and limitOrd Table 48
\( \alpha \) is a notation for one of expressions 8 to 12.

\[
\omega_{k, \gamma}(\beta_1, \beta_2, \ldots, \beta_m) \left[ [\delta] \omega_k[\eta] \right] \left[ [\delta] \omega_{k, \gamma}(\beta_1, \beta_2, \ldots, \beta_m) \right] = [\delta] \omega_k[\eta]
\]

See Table 35 for symbol definitions.

<table>
<thead>
<tr>
<th>X</th>
<th>Condition(s)</th>
<th>LimitTypeInfo</th>
<th>( \alpha.\le(n) )</th>
</tr>
</thead>
<tbody>
<tr>
<td>LEAD</td>
<td>(( \eta \neq 0 ))</td>
<td>drillDown*</td>
<td>drillDownLimitElement(n)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>See Table 38.</td>
</tr>
<tr>
<td>LEBC</td>
<td>various conditions</td>
<td>many options</td>
<td>limitElementCom(n)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>See Table 37.</td>
</tr>
<tr>
<td>LECK</td>
<td>( \kappa = \delta \wedge ) ( \text{sz} = 1 \wedge (\gamma = 0) \wedge \beta_1.isSc \wedge \kappa.isSc )</td>
<td>indexCKsuccParamEq</td>
<td>( b = [\kappa] \omega_k(\beta_1 - 1); ) ( \text{for (i=1; i&lt;n; i++)} ) ( b = \omega_{k-1}.b.lp1 ); Rtn b</td>
</tr>
<tr>
<td>LEDC</td>
<td>( \text{sz} = 0 \wedge (\gamma = 0) \wedge ) ( \kappa.isSc \wedge \delta = 0 )</td>
<td>indexCKsuccUn</td>
<td>( \omega_k[\eta] )</td>
</tr>
<tr>
<td>LEDE</td>
<td>( \kappa.isSc \wedge (\delta &gt; 0) )</td>
<td>indexCKsuccEmbed</td>
<td>( [\delta] \omega_k[\eta] )</td>
</tr>
<tr>
<td>LEEE</td>
<td>( \kappa.isLm \wedge (\delta &gt; 0) )</td>
<td>indexCKlimitEmbed</td>
<td>( \kappa' = \text{incLm}(\delta, n); \text{Rtn } [\delta] \omega_{k'} )</td>
</tr>
</tbody>
</table>

See Table 38 for examples

Table 36: AdmisNormalElement::limitElement cases

<table>
<thead>
<tr>
<th>X</th>
<th>Condition(s)</th>
<th>LimitTypeInfo</th>
<th>( \alpha.\le(n) )</th>
</tr>
</thead>
<tbody>
<tr>
<td>LCAF</td>
<td>( \gamma.\text{isSc} \vee \text{sz} &gt; 1 \vee ((\gamma = 0) \wedge (\text{sz} = 1)) )</td>
<td>param* function*</td>
<td>IterFuncNormalElement::limitElementCom(n)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>See Section 7.2.</td>
</tr>
<tr>
<td>LCBL</td>
<td>( \kappa.isLm \wedge (\gamma = 0) \wedge (\delta = 0) )</td>
<td>indexCKlimit</td>
<td>( \omega_{k.\le(n),\eta}.lp1() )</td>
</tr>
<tr>
<td>LCCI</td>
<td>( \text{sz} = 1 \wedge (\gamma = 0) \wedge \beta_1.isSc \wedge \kappa = 1 \wedge \delta = 0 )</td>
<td>indexCKsuccParam</td>
<td>( b = \omega_k(\beta_1 - 1); ) ( \text{for (i=1; i&lt;n; i++)} ) ( b = \varphi_{b.lp1}(\beta_1 - 1); \text{Rtn b} )</td>
</tr>
<tr>
<td>LCDP</td>
<td>( \text{sz} = 1 \wedge (\gamma = 0) \wedge 1 \wedge \delta &gt; \kappa )</td>
<td>indexCKsuccParam</td>
<td>( b = [\delta] \omega_k(\beta_1 - 1); ) ( \text{for (i=1; i&lt;n; i++)} ) ( b = \omega_{k-1}.b.lp1(\beta_1 - 1); \text{Rtn b} )</td>
</tr>
<tr>
<td>LCEL</td>
<td>( \text{sz} = 1 \wedge (\gamma = 0) \wedge 1 \wedge \beta_1 \text{isSc} \wedge 1 \wedge \delta )</td>
<td>indexCKlimitParamUn</td>
<td>( l = \kappa.\le(n); ) ( [\delta] \omega_{l.eUse(1),\omega_k(\beta_1-1)}.lp1 )</td>
</tr>
</tbody>
</table>

See Table 40 for examples

Table 37: AdmisNormalElement::limitElementCom cases
\( \alpha \) is a notation from expressions 9, 10 or 12.
\[
\alpha = \omega_\kappa[\eta] \quad \alpha = [[\delta]\omega_\kappa[\eta]] \quad |\delta|\omega_\kappa[\eta]
\]
The \( \delta \) parameter is unchanged and not displayed if optional.

See Table 35 for symbol definitions.

<table>
<thead>
<tr>
<th>X</th>
<th>Condition(s)</th>
<th>LimitTypeInfo</th>
<th>( \alpha.\text{le}(n) )</th>
</tr>
</thead>
<tbody>
<tr>
<td>DDAC</td>
<td>( \delta \text{isOne} \land \kappa \text{isOne} )</td>
<td>drillDownLimit, drillDownSucc, drillDownOne, drillDownSuccEmbed, drillDownOneEmbed</td>
<td>( \omega_\kappa[\eta] \text{.limitElement}(n) \text{ or } [[\delta]\omega_\kappa[[\eta] \text{.limitElement}(n)]] )</td>
</tr>
<tr>
<td>DDBO</td>
<td>( \eta \text{.isSc} \land \eta &gt; 1 )</td>
<td>drillDownCKOne</td>
<td>( b = \omega \text{; for (i=1;i&lt;n;i++) b=}\varphi_{b,lp1};\text{Rtn b} )</td>
</tr>
<tr>
<td>DDCO</td>
<td>( \land \kappa == 1 )</td>
<td>drillDownSuccCKOne</td>
<td>see Table 42 for examples</td>
</tr>
</tbody>
</table>

Table 38: AdmisNormalElement::drillDownLimitElement cases

<table>
<thead>
<tr>
<th>X</th>
<th>Condition(s)</th>
<th>LimitTypeInfo</th>
<th>( \alpha.\text{drillDownLimitElementCom}(n) )</th>
</tr>
</thead>
<tbody>
<tr>
<td>DCAL</td>
<td>( \eta \text{.isLm} )</td>
<td>drillDownLimit</td>
<td>( \omega_\kappa[[\eta] \text{.limitElement}(n)] \text{ or } [[\delta]\omega_\kappa[[\eta] \text{.limitElement}(n)] )</td>
</tr>
<tr>
<td>DCBO</td>
<td>( \text{isDdEmb} \land \eta == 1 )</td>
<td>drillDownOneEmbed</td>
<td>( b = \omega \text{; for (i=1;i&lt;n;i++) b=}\varphi_{b,lp1};\text{Rtn b} )</td>
</tr>
<tr>
<td>DCCS</td>
<td>( \eta \text{.isSc} &gt; 1 \land \text{isDdEmb} )</td>
<td>drillDownSuccEmbed</td>
<td>( b = [[\delta]\omega_\kappa[[\eta] - 1]];\text{for (i=1;i&lt;n;i++) b=}\varphi_{b,lp1};\text{Rtn b} )</td>
</tr>
<tr>
<td>DCDO</td>
<td>( (\eta == 1)\land \text{isDdEmb} \land (\kappa &gt; 1) )</td>
<td>drillDownOne</td>
<td>( b = [[\delta]\omega_{\kappa-1,\text{lp1}}];\text{Rtn b} )</td>
</tr>
<tr>
<td>DCES</td>
<td>( !\text{isDdEmb} \land (\eta \text{.isSc} &gt; 1) \land (\kappa &gt; 1) )</td>
<td>drillDownSucc</td>
<td>( b = \omega_{\kappa \eta - 1};\text{for (i=1;i&lt;n;i++) b=}\varphi_{b,lp1};\text{Rtn b} )</td>
</tr>
</tbody>
</table>

Table 39: AdmisNormalElement::drillDownLimitElementCom cases
Table 40: AdmisNormalElement::limitElement exit codes

<table>
<thead>
<tr>
<th>X</th>
<th>UpX</th>
<th>Ordinal</th>
<th>limitElement</th>
</tr>
</thead>
<tbody>
<tr>
<td>&amp;X</td>
<td>LEAD</td>
<td>LEAD</td>
<td>( \omega_1 )</td>
</tr>
<tr>
<td>DDCO</td>
<td>LEBC</td>
<td>LEBC</td>
<td>( \omega_2 )</td>
</tr>
<tr>
<td>LCBEL</td>
<td>LEBC</td>
<td>LEBC</td>
<td>( \omega_3 )</td>
</tr>
<tr>
<td>LCBEL</td>
<td>LEBC</td>
<td>LEBC</td>
<td>( \omega_4 )</td>
</tr>
<tr>
<td>LCBEL</td>
<td>LEBC</td>
<td>LEBC</td>
<td>( \omega_5 )</td>
</tr>
<tr>
<td>LCBEL</td>
<td>LEBC</td>
<td>LEBC</td>
<td>( \omega_6 )</td>
</tr>
<tr>
<td>LCBEL</td>
<td>LEBC</td>
<td>LEBC</td>
<td>( \omega_7 )</td>
</tr>
<tr>
<td>LCBEL</td>
<td>LEBC</td>
<td>LEBC</td>
<td>( \omega_8 )</td>
</tr>
</tbody>
</table>

&X is an exit code (see Table 36). UpX is a higher level exit code from a calling routine.
X is an exit code (see Table 37). UpX is a higher level exit code from a calling routine.

<table>
<thead>
<tr>
<th>X</th>
<th>UpX</th>
<th>Ordinal</th>
<th>limitElement</th>
</tr>
</thead>
<tbody>
<tr>
<td>II</td>
<td>LCAF</td>
<td>$\omega_{12,4}(1)$</td>
<td>$\omega_{12,4} + 1$</td>
</tr>
<tr>
<td>II</td>
<td>LCAF</td>
<td>$[3] \omega_{12,4}(1)$</td>
<td>$[3] \omega_{12,4} + 1$</td>
</tr>
<tr>
<td>LCBL</td>
<td>LEBC</td>
<td>$\omega_1$</td>
<td>$\omega_2$</td>
</tr>
<tr>
<td>LCBL</td>
<td>LEBC</td>
<td>$\omega_{\omega_1}$</td>
<td>$\omega_{\omega_2} + 1$</td>
</tr>
<tr>
<td>LCCI</td>
<td>LEBC</td>
<td>$\omega_1(12)$</td>
<td>$\omega_1(11)$</td>
</tr>
<tr>
<td>LCDP</td>
<td>LEBC</td>
<td>$\omega_5(12)$</td>
<td>$\omega_5(11)$</td>
</tr>
<tr>
<td>LCEL</td>
<td>LEBC</td>
<td>$[5] \omega_6([5] \omega_1(12)+1)$</td>
<td>$[5] \omega_6([5] \omega_1(11)+1)$</td>
</tr>
</tbody>
</table>

Table 41: AdmisNormalElement::limitElementCom exit codes

X is an exit code (see Table 38). UpX is a higher level exit code from a calling routine.

<table>
<thead>
<tr>
<th>X</th>
<th>UpX</th>
<th>Ordinal</th>
<th>limitElement</th>
</tr>
</thead>
<tbody>
<tr>
<td>DCDO</td>
<td>DDAC</td>
<td>$\omega_2[1]$</td>
<td>$\omega_1$</td>
</tr>
<tr>
<td>DCAL</td>
<td>DDAC</td>
<td>$\omega_2[\omega_1]$</td>
<td>$\omega_2[1]$</td>
</tr>
<tr>
<td>DCCS</td>
<td>DDAC</td>
<td>$[12] \omega_{\omega_1+1}[\omega + 3]$</td>
<td>$[12] \omega_{\omega_1+1}[\omega + 2]$</td>
</tr>
<tr>
<td>DDBO</td>
<td>LEAD</td>
<td>$\omega_1[1]$</td>
<td>$\omega$</td>
</tr>
<tr>
<td>DDCO</td>
<td>LEAD</td>
<td>$\omega_1[\omega + 1]$</td>
<td>$\omega_1[\omega]$</td>
</tr>
<tr>
<td>DDCO</td>
<td>LEAD</td>
<td>$\omega_1[13]$</td>
<td>$\omega_1[12]$</td>
</tr>
</tbody>
</table>

Table 42: AdmisNormalElement::drillDownLimitElement exit codes
X is an exit code (see Table 39). UpX is a higher level exit code from a calling routine.

<table>
<thead>
<tr>
<th>X</th>
<th>UpX</th>
<th>Ordinal</th>
<th>limitElement</th>
</tr>
</thead>
<tbody>
<tr>
<td>DCAL</td>
<td>LEAD</td>
<td>Ω6ω3</td>
<td>5 [Ω6ω31]</td>
</tr>
<tr>
<td>DCAL</td>
<td>LEAD</td>
<td>Ω6ω1</td>
<td>5 [Ω6ω31]</td>
</tr>
<tr>
<td>DCAL</td>
<td>LEAD</td>
<td>Ω6ω1</td>
<td>5 [Ω6ω31]</td>
</tr>
<tr>
<td>DCBO</td>
<td>LEAD</td>
<td>Ω4ω1</td>
<td>4 [Ω4ω11]</td>
</tr>
<tr>
<td>DCBO</td>
<td>LEAD</td>
<td>Ω4ω1</td>
<td>4 [Ω4ω11]</td>
</tr>
<tr>
<td>DCBO</td>
<td>LEAD</td>
<td>Ω4ω1</td>
<td>4 [Ω4ω11]</td>
</tr>
<tr>
<td>DCCS</td>
<td>LEAD</td>
<td>Ω3ω3</td>
<td>3 [Ω3ω31]</td>
</tr>
<tr>
<td>DCCS</td>
<td>LEAD</td>
<td>Ω3ω3</td>
<td>3 [Ω3ω31]</td>
</tr>
<tr>
<td>DCCS</td>
<td>LEAD</td>
<td>Ω3ω3</td>
<td>3 [Ω3ω31]</td>
</tr>
<tr>
<td>DCDO</td>
<td>LEAD</td>
<td>Ω8ω7</td>
<td>Ω8ω7ω7</td>
</tr>
<tr>
<td>DCDO</td>
<td>LEAD</td>
<td>Ω8ω7</td>
<td>Ω8ω7ω7</td>
</tr>
<tr>
<td>DCDO</td>
<td>LEAD</td>
<td>Ω8ω7</td>
<td>Ω8ω7ω7</td>
</tr>
<tr>
<td>DCES</td>
<td>LEAD</td>
<td>Ω3ω3</td>
<td>3 [Ω3ω31]</td>
</tr>
<tr>
<td>DCES</td>
<td>LEAD</td>
<td>Ω3ω3</td>
<td>3 [Ω3ω31]</td>
</tr>
<tr>
<td>DCES</td>
<td>LEAD</td>
<td>Ω3ω3</td>
<td>3 [Ω3ω31]</td>
</tr>
</tbody>
</table>

Table 43: AdmisNormalElement::drillDownLimitElementCom exit codes
The following is at code level `cantorCodeLevel` (Section 4) and above.

<table>
<thead>
<tr>
<th>LimitTypeInfo</th>
<th>Ordinal</th>
<th>limitElement</th>
</tr>
</thead>
<tbody>
<tr>
<td>paramSucc</td>
<td>(\omega^{\omega+12})</td>
<td>(\omega^{\omega+11})</td>
</tr>
<tr>
<td>paramLimit</td>
<td>(\varphi(0,0))</td>
<td>(\varepsilon_0)</td>
</tr>
<tr>
<td>paramSuccZero</td>
<td>(\varphi(\omega+12,0,0))</td>
<td>(\varphi(\omega+11,1,0))</td>
</tr>
<tr>
<td>paramSucc</td>
<td>(\varphi(\omega+2,5,3))</td>
<td>(\varphi(\omega+2,5,2))</td>
</tr>
<tr>
<td>paramNxtLimit</td>
<td>(\varphi(\varepsilon_0,0,0,33))</td>
<td>(\varphi(\omega,3,\varphi(\varepsilon_0,0,0,32)+1,0,33))</td>
</tr>
<tr>
<td>functionSucc</td>
<td>(\varphi_{11})</td>
<td>(\varphi_{10}(\varphi_{10}+1))</td>
</tr>
<tr>
<td>functionNxtSucc</td>
<td>(\varphi_5(4))</td>
<td>(\varphi_5(3,1,0)+1)</td>
</tr>
<tr>
<td>functionLimit</td>
<td>(\varphi_{16}(4,0,0))</td>
<td>(\varphi_{16}(3,\varphi_5(3,1,0)+1,0)+1)</td>
</tr>
<tr>
<td>functionNxtLimit</td>
<td>(\varphi_{16}(15))</td>
<td>(\varphi_{16}(14)+1)</td>
</tr>
<tr>
<td>drillDownLimit</td>
<td>([2] \omega_3[2])</td>
<td>([2] \omega_4[1])</td>
</tr>
<tr>
<td>drillDownSucc</td>
<td>([2] \omega_5[5])</td>
<td>([2] \omega_6[4])</td>
</tr>
<tr>
<td>drillDownSuccCKOne</td>
<td>(\omega_2[2])</td>
<td>(\omega_3[1])</td>
</tr>
<tr>
<td>drillDownSuccEmbed</td>
<td>([3] \omega_3[7])</td>
<td>([3] \omega_4[6])</td>
</tr>
<tr>
<td>drillDownOne</td>
<td>([3] \omega_3[5]+1)</td>
<td>([3] \omega_4+4)</td>
</tr>
<tr>
<td>drillDownOneEmbed</td>
<td>([3] \omega_5[1])</td>
<td>([3] \omega_5+5)</td>
</tr>
<tr>
<td>drillDownCKOne</td>
<td>(\omega_1[1])</td>
<td>(\omega_2[2])</td>
</tr>
<tr>
<td>indexCKlimit</td>
<td>(\omega_1^2)</td>
<td>(\omega_2^2)</td>
</tr>
<tr>
<td>indexCKsuccParam</td>
<td>([3] \omega_1[7])</td>
<td>([3] \omega_2[6])</td>
</tr>
<tr>
<td>indexCKsuccParamEq</td>
<td>([12] \omega_1[7])</td>
<td>([12] \omega_2[6])</td>
</tr>
<tr>
<td>indexCKsuccEmbed</td>
<td>([2] \omega_2[1])</td>
<td>([2] \omega_3[2])</td>
</tr>
<tr>
<td>indexCKsuccUn</td>
<td>(\omega_3+6)</td>
<td>(\omega_4+6)</td>
</tr>
<tr>
<td>indexCKlimitParamUn</td>
<td>([\omega^2+1] \omega_2[5])</td>
<td>([\omega^2+1] \omega_2[5]+1,\omega_3[4]+1)</td>
</tr>
<tr>
<td>indexCKlimitEmbed</td>
<td>([12] \omega_2[12])</td>
<td>([12] \omega_2[12])</td>
</tr>
</tbody>
</table>

Table 44: `limitElement` and LimitTypeInfo examples through `admisCodeLevel`
<table>
<thead>
<tr>
<th>AdmisLevOrdinal</th>
<th>1</th>
<th>2</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>ω</td>
<td>ω₁</td>
<td>ω₂</td>
<td>ω₃</td>
</tr>
<tr>
<td>ϕ(ω₁, 1, 0, 0)</td>
<td>ϕ(ω₁, 1, 0, 0)</td>
<td>ϕ(ω₁, 0, ϕ(ω₁, 1, 0, 0) + 1, 0)</td>
<td>ϕ(ω₁, 0, ϕ(ω₁, 0, ϕ(ω₁, 1, 0, 0) + 1, 0) + 1, 0)</td>
</tr>
<tr>
<td>ϕ(ω₁ + 1, 1)</td>
<td>ϕ(ω₁ + 1, 0)</td>
<td>ϕ(ω₁, ϕ(ω₁ + 1, 0) + 1)</td>
<td>ϕ(ω₁, ϕ(ω₁, ϕ(ω₁ + 1, 0) + 1) + 1)</td>
</tr>
<tr>
<td>ω₁[1]</td>
<td>ω</td>
<td>ϕω</td>
<td>ϕω₊₁</td>
</tr>
</tbody>
</table>

| ω₁(1, 1)        | ω₁(1, 0) | ω₁(ω₁(1, 0) + 1) | ω₁(ω₁(ω₁(1, 0) + 1) + 1) |
| ω₁(0, 1)        | ω₁(0, 0) | ω₁(ω₁(0, 0) + 1, 1) | ω₁(ω₁(ω₁(0, 0) + 1, 1) + 1, 1) |
| ω₁(1, 0)        | ω₁(1, 0, 1) | ω₁(1, 0, ω₁(1, 0, 1) + 1) | ω₁(1, 0, ω₁(1, 0, ω₁(1, 0, 1) + 1) + 1) |
| ω₁(ε₀, 0)       | ω₁(ε₀, 0) | ω₁(ω₁(ε₀, 0) + 1) | ω₁(ω₁(ω₁(ε₀, 0) + 1) + 1) |
| ω₁(ε₀, 1)       | ω₁(ω₁(ε₀, 0, 1) + 1) | ω₁(ω₁(ω₁(ε₀, 0, 1) + 1) + 1) | ω₁(ω₁(ω₁(ω₁(ε₀, 0, 1) + 1) + 1) + 1) |
| ω₁(ε₀, 1)       | ω₁(ω₁(ε₀, 0, 1) + 1) | ω₁(ω₁(ω₁(ε₀, 0, 1) + 1) + 1) | ω₁(ω₁(ω₁(ω₁(ε₀, 0, 1) + 1) + 1) + 1) |
| ω₁(ε₀, 1)       | ω₁(ω₁(ε₀, 0, 1) + 1) | ω₁(ω₁(ω₁(ε₀, 0, 1) + 1) + 1) | ω₁(ω₁(ω₁(ω₁(ε₀, 0, 1) + 1) + 1) + 1) |

Table 45: AdmisLevOrdinal::limitElement examples part 1.
<table>
<thead>
<tr>
<th>AdmisLevOrdinal</th>
<th>limitElement</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\omega_{1,1}(1)$</td>
<td>$\omega_{1,1} + 1$</td>
</tr>
<tr>
<td>$\omega_{2,1}(1)$</td>
<td>$\omega_{2,1} + 1$</td>
</tr>
<tr>
<td>$\omega_{1,3}(1)$</td>
<td>$\omega_{1,3} + 1$</td>
</tr>
<tr>
<td>$\omega_{1,3}(5)$</td>
<td>$\omega_{1,3}(4) + 1$</td>
</tr>
<tr>
<td>$\omega_{1,1}(1,0)$</td>
<td>$\omega_{1,1}(1)$</td>
</tr>
<tr>
<td>$\omega_{1,3}(1,0)$</td>
<td>$\omega_{1,3}(1)$</td>
</tr>
<tr>
<td>$\omega_{1,3}(5,0)$</td>
<td>$\omega_{1,3}(4,1)$</td>
</tr>
<tr>
<td>$\omega_{1,1}(1,0,0)$</td>
<td>$\omega_{1,1}(1,0)$</td>
</tr>
<tr>
<td>$\omega_{1,3}(1,0,0)$</td>
<td>$\omega_{1,3}(1,0)$</td>
</tr>
<tr>
<td>$\omega_{1,3}(2,0,0)$</td>
<td>$\omega_{1,3}(1,1,0)$</td>
</tr>
<tr>
<td>$\omega_{1,0,0}(1)$</td>
<td>$\omega_1,\omega(\omega_{1,0} + 1)$</td>
</tr>
<tr>
<td>$\omega_{1,0,0+1}(1)$</td>
<td>$\omega_{1,0} + 1,0$</td>
</tr>
<tr>
<td>$\omega_{1,0,0+\omega}(3)$</td>
<td>$\omega_{1,0} + 1(\omega_{1,0} + \omega(2) + 1)$</td>
</tr>
<tr>
<td>$\omega_{1,1}(1)$</td>
<td>$\omega_{1,1} + 1$</td>
</tr>
<tr>
<td>$\omega_{1,0,0}(1)$</td>
<td>$\omega_{1,0} + 1$</td>
</tr>
<tr>
<td>$\omega_{1,0,0+\omega}(1)$</td>
<td>$\omega_{1,0} + 1(\omega_{1,0} + \omega(2) + 1)$</td>
</tr>
<tr>
<td>$\omega_{1,0,0}(5)$</td>
<td>$\omega_{1,0} + 1(\omega_{1,0} + \omega(4) + 1)$</td>
</tr>
<tr>
<td>$\omega_{1,0,0+\omega}(5)$</td>
<td>$\omega_{1,0} + 1(\omega_{1,0} + \omega(4) + 1)$</td>
</tr>
<tr>
<td>$\omega_{1,0}(1)$</td>
<td>$\omega_{1,0}(1)$</td>
</tr>
<tr>
<td>$\omega_{1,0+1}(1)$</td>
<td>$\omega_{1,0} + 1$</td>
</tr>
<tr>
<td>$\omega_{1,0+\omega}(1)$</td>
<td>$\omega_{1,0} + \omega(1)$</td>
</tr>
<tr>
<td>$\omega_{1,0}(5)$</td>
<td>$\omega_{1,0}(5)$</td>
</tr>
<tr>
<td>$\omega_{1,0+\omega}(5)$</td>
<td>$\omega_{1,0} + \omega(5)$</td>
</tr>
<tr>
<td>$\omega_{1,0+\omega+1}(1)$</td>
<td>$\omega_{1,0} + \omega + 1$</td>
</tr>
<tr>
<td>$\omega_{1,0+\omega+\omega}(1)$</td>
<td>$\omega_{1,0} + \omega + \omega$</td>
</tr>
</tbody>
</table>

Table 46: AdmisLevOrdinal::limitElement examples part 2.
<table>
<thead>
<tr>
<th>AdminLevOrdinal</th>
<th>1</th>
<th>2</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\omega^{1+1}$</td>
<td>$\omega_1$</td>
<td>$\omega_1 + \omega_1[1]$</td>
<td>$(\omega_1)^2$</td>
</tr>
<tr>
<td>$(\omega_2)$</td>
<td>$\omega_1 + \omega_1[1]$</td>
<td>$\omega_1 + \omega_1[2]$</td>
<td>$\omega_1 + \omega_1[3]$</td>
</tr>
<tr>
<td>$\omega_{\varepsilon_0}(1)$</td>
<td>$\omega_2^{\omega_2[1]+1}$</td>
<td>$\omega_2^{\omega_2,\omega_{\varepsilon_0}[1]+1}$</td>
<td>$\omega_2^{\omega_2,\omega_{\varepsilon_0}[1]+1}$</td>
</tr>
<tr>
<td>$\omega_{\varepsilon_0}(2)$</td>
<td>$\omega_2^{\omega_2[1]+1}$</td>
<td>$\omega_2^{\omega_{\varepsilon_0}[1]+1}$</td>
<td>$\omega_2^{\omega_{\varepsilon_0}[1]+1}$</td>
</tr>
<tr>
<td>$[\omega + 1]</td>
<td>\omega_{\varepsilon_0}$</td>
<td>$[\omega + 1]</td>
<td>\omega_{\varepsilon_0}[1]+1$</td>
</tr>
<tr>
<td>$[\omega + 1]</td>
<td>\omega_{\varepsilon_0}[1]+1$</td>
<td>$[\omega + 1]</td>
<td>\omega_{\varepsilon_0}[1]+1$</td>
</tr>
</tbody>
</table>

Table 47: AdminLevOrdinal::limitElement examples part 3.
9.3 isValidLimitOrdParam member function

For admissible level ordinals, limitElement must be supplemented with limitOrd that accepts Ordinals up to a given size as input as discussed in Section 8.3. Ordinal member function isValidLimitOrdParam returns true if its single argument is a valid parameter to limitOrd when called from the object isValidLimitOrdParam is called from. (Although this is an Ordinal member function it is only required at classes AdmisLevOrdinal and higher.) It uses limitType and maxLimitType as discussed below if no \([\[\delta\]\] prefix is involved. Otherwise it must also use embedType and the relative sizes of the object it is called from and its parameter to determine if it is a legal argument. The problem is that \([\[\delta\]\] \(\omega_\delta\) (with \(\delta\) a successor) creates an ordinal \(< \omega_\delta\) that still must accept smaller ordinals such as \([\[\delta\]\] \(\omega_\delta\)[1]) as a parameters. \((([\delta]\omega_\delta[1]).\maxLimitType == ([\delta]\omega_\delta).\limitType.)\) When limitType of the object limitOrd is called from equals maxLimitType of the argument and embedType is non null the argument is valid if it is less than the object limitOrd is called from.

9.4 limitInfo, limitType and embedType member functions

Ordinarily these routines are used indirectly by calling isValidLimitOrdParam described in the previous section. They are introduced in this class because they are not used in a system with only notations for base classes. However they are defined in these base classes. Only limitInfo has a version for class AdmisNormalElement. It does the bulk of the work computing the limitType value and the enum LimitTypeInfo used in routines limitElement and limitOrd. The algorithm for this is outlined in see Table 34 for the description through admisCodeLevel and Table 52 for all classes discussed in this document along with examples in Table 53. Clicking on an entry in the LimitInfoType will take you do the example line in the second table.

limitType and embedType both return an OrdinalImpl. The limitType of an ordinal is the limitType of the least significant term.

9.5 maxLimitType member function

maxLimitType is the maximum of limitType for all ordinals \(\leq\) the ordinal represented by the object maxLimitType is called from. AdmisNormalElement::maxLimitTypes calls IterFuncNormalElement::maxLimitType for the maximum of all parameters except those unique to class AdmisLevOrdinal. Next, the type of \(\omega_\kappa\) is computed and the maximum of these two values is taken. Finally, the effect of the \(\delta\) and \(\eta\) parameters are taken into account. \(\delta\) puts an upper bound on maxLimitType. If \(\delta\) is zero a nonzero \(\eta\) reduces the value of maxLimitType by 1 if it is a successor.

9.6 limitOrd member function

AdmisNormalElement::limitOrd extends the idea of limitElement as indirectly enumerating all smaller ordinals. It does this in a limited way for ordinals that are not recursive by using ordinal notations (including those yet to be defined) as arguments in place of the integer arguments of limitElement. By defining recursive operations on an
incomplete domain we can retain something of the flavor of `limitElement` since: \( \alpha = \bigcup \beta: \alpha.isValidLimitOrdParam(\beta) \alpha.limitOrd(\beta) \) and \( \alpha.isValidLimitOrdParam(\beta) \rightarrow \beta < \alpha. \)

Table 48 gives the logic of `AdmisNormalElement::limitOrd` along with the type of limit designated by `LimitInfoType` and the exit code used in debugging. Table 49 gives examples for each exit code. Table 50 gives values of `enum LimitInfoType` used by `limitOrd` in a `case` statement along with examples. Usually `limitOrd` is simpler than `limitElement` because it adds its argument as a \([\eta]\) or \([[[\eta]]]\) parameter to an existing parameter. Sometimes it must also decrement a successor ordinal and incorporate this in the result. The selection of which parameter(s) to modify is largely determined by a `case` statement on `LimitInfoType`. There are some additional examples in Table 51.

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

One complication occurs when \( \kappa \) is the least significant non zero parameter and \( \delta \) is nonzero. Since \( \delta \) cannot be a limit it must be less than \( \kappa \) in this case. The value returned by `limitOrd` must not have \( \kappa > \delta \). This is insured with routine `leUse`\(^{48}\) which in turn calls `AdmisNormalElement::increasingLimit`\(^{49}\).

\(^{48}\) `AdmisNormalElement::leUse(lo)` is called with \( lo = \kappa.limitOrd(ord) \) as an inargument. It calls `increasingLimit` (see Note 49) to insure a valid output that will be strictly increasing for increasing inputs.

\(^{49}\) `AdmisNormalElement::increasingLimit(effDelta, limitElt)` computes a value \( \geq effDelta \) that will be strictly increasing for increasing values of `limitElt`. This insures `limitElement` and `limitOrd` will not violate the \( \delta \) constraint on \( \kappa \) and will produce increasing outputs from increasing inputs. Terms may be added to the output from `effDelta` to insure this. All terms in `effDelta` that are in `limitElt` or less than terms in `limitElt` excluding the last term in `limitElt` are ignored. The remainder are added to `limitElt`. 74
\( \alpha \) is a notation from expressions 8 to 12.

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

See Table 35 for symbol definitions.

<table>
<thead>
<tr>
<th>X</th>
<th>Condition(s)</th>
<th>Info</th>
<th>( \alpha . \text{limitOrd}(\zeta) )</th>
</tr>
</thead>
<tbody>
<tr>
<td>LOA</td>
<td>( \eta . \text{isLm} )</td>
<td>drillDownLimit</td>
<td>( \omega_{\kappa}[\eta . \text{lo}(\zeta)] ) or ( [[\delta]]\omega_{\kappa}[\eta . \text{lo}(\zeta)] )</td>
</tr>
<tr>
<td>LOB</td>
<td>least significant ( \beta_i ) or ( \gamma ) is limit</td>
<td>paramLimit paramNxtLimit functionLimit functionNxtLimit</td>
<td>( \alpha . \text{IfNe}.\text{lo}(\zeta) )</td>
</tr>
<tr>
<td>LOC</td>
<td>(sz==0) &amp; (\gamma == 0) &amp; (delta == 0) &amp; k . isLm</td>
<td>indexCKLimit</td>
<td>( \omega_{\kappa}.\text{lo}(\zeta) )</td>
</tr>
<tr>
<td>LOD</td>
<td>(sz==0) &amp; (\gamma == 0) &amp; \delta . isSc &amp; k . isLm</td>
<td>indexCKLimitEmbed</td>
<td>( [[\delta]]\omega_{\text{leUse}}(\text{lo}(\zeta)) )</td>
</tr>
<tr>
<td>LOE</td>
<td>(sz==1) &amp; (\gamma == 0) &amp; \beta_1 . isSc &amp; k . isLm &amp; \delta . isSc</td>
<td>indexCKLimitParamUn</td>
<td>( 1 = k . \text{lo}(\zeta); \quad [[\delta]]\omega_{\text{leUse}(1)}(\omega_{\kappa}(\beta_1 - 1) . l) )</td>
</tr>
<tr>
<td>LOF</td>
<td>(sz==0) &amp; (\gamma == 0) &amp; \kappa . isSc &amp; \delta . isSc</td>
<td>indexCKsuccEmbed</td>
<td>( [[\delta]]\omega_{\kappa}[\zeta] )</td>
</tr>
<tr>
<td>LOF</td>
<td>(sz==0) &amp; (\gamma == 0) &amp; \kappa . isSc &amp; (\delta == 0)</td>
<td>indexCKsuccUn</td>
<td>( \omega_{\kappa}[\zeta] )</td>
</tr>
</tbody>
</table>

Table 48: AdmisNormalElement::limitOrdCom cases
X is an exit code (see Table 48). UpX is a higher level exit code from a calling routine.

<table>
<thead>
<tr>
<th>X</th>
<th>UpX</th>
<th>Ordinal</th>
<th>limitOrd</th>
</tr>
</thead>
<tbody>
<tr>
<td>LOA</td>
<td>$\omega_{12}[\omega_{11}]$</td>
<td>$\omega_{12}[\omega_{11} + 5]$</td>
<td>$\omega_{12}[\omega_{11} \varphi(4, 0, 0) + 12]$</td>
</tr>
<tr>
<td>OFB</td>
<td>$[\omega + 3]<a href="%5Comega_%7B11%7D">\omega_3 + 2</a>$</td>
<td>$[\omega + 3][\omega_3 + 2][\omega_{11} + 5]$</td>
<td>$[\omega + 3][\omega_3 + 2][\omega_{11} \varphi(4, 0, 0) + 12] + 1$</td>
</tr>
<tr>
<td>OFA</td>
<td>$\omega_{9}(\omega_{11}, 1)$</td>
<td>$\omega_{9}(\omega_{11} + 5) + 1, \omega_{11} + 1$</td>
<td>$\omega_{9}(\omega_{11} \varphi(4, 0, 0) + 12) + 1, \omega_{11} + 1$</td>
</tr>
<tr>
<td>FOB</td>
<td>$[5][\omega_8, \omega_7]$</td>
<td>$[5][\omega_8, \omega_7][\omega_{12} + 1]$</td>
<td>$[5][\omega_8, \omega_7][\omega_{12} \varphi(4, 0, 0) + 13] + 1$</td>
</tr>
<tr>
<td>FOC</td>
<td>$\omega_{12}[\omega_{11} + 1]$</td>
<td>$\omega_{12}[\omega_{11} + 5] + 1, \omega_{12}[\omega_{11} + 1]$</td>
<td>$\omega_{12}[\omega_{11} \varphi(4, 0, 0) + 15] + 1$</td>
</tr>
<tr>
<td>LOC</td>
<td>$\omega_\omega_{\omega_5 + 12}$</td>
<td>$\omega_\omega_{\omega_5 + 12} + \omega_{5}$</td>
<td>$\omega_\omega_{\omega_5 + 12} \varphi(4, 0, 0) + 12$</td>
</tr>
<tr>
<td>LOD</td>
<td>$[\omega + 3][\omega_\omega_{\omega_5 + 12}] + \omega_{5}$</td>
<td>$[\omega + 3][\omega_\omega_{\omega_5 + 12} + 5]$</td>
<td>$[\omega + 3][\omega_\omega_{\omega_5 + 12} \varphi(4, 0, 0) + 12] + \omega_{5}$</td>
</tr>
<tr>
<td>LOE</td>
<td>$\omega_{\omega_{\omega_5 + 12}}(12)$</td>
<td>$\omega_{\omega_{\omega_5 + 12} + 5}, \omega_{\omega_{\omega_5 + 12}(11) + 1}$</td>
<td>$\omega_{\omega_{\omega_5 + 12} \varphi(4, 0, 0) + 12}, \omega_{\omega_{\omega_5 + 12}(11) + 1}$</td>
</tr>
<tr>
<td>LOE</td>
<td>$[19][\omega_\omega_{\omega_5 + 12}(12)]$</td>
<td>$[19][\omega_\omega_{\omega_5 + 12} + 5], [19][\omega_\omega_{\omega_5 + 12}(11) + 1]$</td>
<td>$[19][\omega_\omega_{\omega_5 + 12} \varphi(4, 0, 0) + 12], [19][\omega_\omega_{\omega_5 + 12}(11) + 1]$</td>
</tr>
<tr>
<td>LOF</td>
<td>$[\omega + 15][\omega_\omega_{\omega_5 + 20}]$</td>
<td>$[\omega + 15][\omega_\omega_{\omega_5 + 20} + 5] + \omega_{5}$</td>
<td>$[\omega + 15][\omega_\omega_{\omega_5 + 20} \varphi(4, 0, 0) + 12]$</td>
</tr>
<tr>
<td>LOF</td>
<td>$\omega_\omega_{\omega_5 + 20}$</td>
<td>$\omega_\omega_{\omega_5 + 20} + 5$</td>
<td>$\omega_\omega_{\omega_5 + 20} \varphi(4, 0, 0) + 12$</td>
</tr>
</tbody>
</table>

Table 49: AdmisNormalElement::limitOrdCom exit codes
\[ \beta_i \text{ defined in: } 7, 8, 11, \gamma \text{ defined in: } 7, 8, 11, \eta \text{ defined in: } 9, 10, 12, \kappa \text{ defined in: } 8, 9, 10, 11, 12, \delta \text{ defined in: } 10, 11, 12. \]

<table>
<thead>
<tr>
<th>LimitTypeInfo</th>
<th>Ordinal</th>
<th>limitType</th>
<th>Ordinal</th>
<th>limitType</th>
</tr>
</thead>
<tbody>
<tr>
<td>The following is at code level finiteFuncCodeLevel (Section 6) and above.</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>paramLimit ( \varphi(\omega,0) )</td>
<td>1</td>
<td>[12] ( \omega_\omega(\omega_4) )</td>
<td>5</td>
<td></td>
</tr>
<tr>
<td>paramNxtLimit ( \varphi(\varepsilon_0,0,0,33) )</td>
<td>1</td>
<td>( \omega_{12}(\omega_{11},0,0,3) )</td>
<td>12</td>
<td></td>
</tr>
<tr>
<td>The following is at code level iterFuncCodeLevel (Section 7) and above.</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>functionLimit ( \varphi(4,0,0,0) )</td>
<td>1</td>
<td>[20] ( \omega_{\omega+12,\omega+10} )</td>
<td>20</td>
<td></td>
</tr>
<tr>
<td>functionNxtLimit ( \varphi(15) )</td>
<td>1</td>
<td>( \omega^{\omega+1} \omega^{\omega +3}(\omega + 5) )</td>
<td>( \omega^\omega )</td>
<td></td>
</tr>
<tr>
<td>The following is at code level admisCodeLevel (Section 9) and above.</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>drillDownLimit ( [2][\omega_3] )</td>
<td>1</td>
<td>( \omega_{\omega+1}[\omega_{12}] )</td>
<td>13</td>
<td></td>
</tr>
<tr>
<td>indexCKlimit ( \omega_\omega )</td>
<td>1</td>
<td>( \omega_\omega )</td>
<td>( \omega_\omega )</td>
<td></td>
</tr>
<tr>
<td>indexCKsuccEmbed ( [2][\omega_2] )</td>
<td>3</td>
<td>[3] ( \omega_{\omega+4} )</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>indexCKsuccUn ( \omega_{\omega+6} )</td>
<td>6</td>
<td>( \omega_{\omega+6} )</td>
<td>( \omega_{\omega+6} )</td>
<td></td>
</tr>
<tr>
<td>indexCKlimitParamUn ( [\omega^2 + 1]<a href="5">\omega_\omega</a> )</td>
<td>1</td>
<td>[9]<a href="3">\omega specified in ( \omega_{12} )</a></td>
<td>13</td>
<td></td>
</tr>
<tr>
<td>indexCKlimitEmbed ( [12][\omega_\omega] )</td>
<td>1</td>
<td>( \varphi(\omega_{\omega+12} + 1,0,0,1) \omega_{\omega+12} \omega_{\omega+1} )</td>
<td>( \omega_{20} + 1 )</td>
<td></td>
</tr>
</tbody>
</table>

Table 50: limitOrd and LimitTypeInfo examples through admisCodeLevel

### 9.7 fixedPoint member function

`AdmisLevOrdinal::fixedPoint` is used by `admisLevelFunctional` to create an instance of an `AdmisLevOrdinal` (expressions 8 to 12) that is the simplest representation of the ordinal. It is not intended for direct use but it is documented here because of the importance of the algorithm. The routine has the following parameters.

- The admissible ordinal index or \( \kappa \).
- The function level or \( \gamma \).
- An index specifying the largest parameter of the ordinal notation being constructed. If the largest parameter is the function level the index has the value `iterMaxParam` defined as \(-1\).
- The function parameters (a `NULL` terminated array of pointers to `Ordinals`) or just `NULL` if there are none. These are the \( \beta_j \) from a term in Expression 7.
- An instance of class `embedding` that contains the value of \( \delta \).

This function determines if the parameter, at the specified index, is a fixed point for an `AdmisLevOrdinal` created with the specified parameters. If so, `true` is returned and otherwise `false`. The routine that calls this routine selects the largest parameter from the function level (\( \gamma \)) and the array of `Ordinal` pointers (\( \beta_j \)) and indicates this in the index parameter. Note no legal value of \( \eta \) and no countable value of the ordinal represented by \( \delta \) (which is always countable in this implementation) can be a fixed point. The calling routine insures that less significant parameters are 0 and the above conditions that preclude a fixed point are not met. This routine is called only if all these checks are passed.
<table>
<thead>
<tr>
<th>parameter</th>
<th>limitOrd</th>
<th>limitOrd</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\omega_1$</td>
<td>$\omega_1 \mid [\omega]$</td>
<td>$\omega_1 \mid \varepsilon_0$</td>
</tr>
<tr>
<td>$\omega_2$</td>
<td>$\omega_2 \mid [\omega]$</td>
<td>$\omega_2 \mid \varepsilon_0$</td>
</tr>
<tr>
<td>$\omega_1$</td>
<td>$\omega_1 \mid [\omega]$</td>
<td>$\omega_1 \mid \varepsilon_0$</td>
</tr>
<tr>
<td>$\omega_{\omega_1+1}(\omega_{\omega_1+1})$</td>
<td>$\omega_{\omega_1+1}(\omega_{\omega_1+1}[\omega] + 1)$</td>
<td>$\omega_{\omega_1+1}(\omega_{\omega_1+1}[\varepsilon_0] + 1)$</td>
</tr>
<tr>
<td>$\omega_{\omega_1+1}$</td>
<td>$\omega_{\omega_1+1}[\omega]$</td>
<td>$\omega_{\omega_1+1}[\varepsilon_0]$</td>
</tr>
<tr>
<td>$\omega_1 \omega_1$</td>
<td>$\omega_1 \omega_1 [\omega] + 1$</td>
<td>$\omega_1 \omega_1 [\varepsilon_0] + 1$</td>
</tr>
<tr>
<td>$\omega_{100}$</td>
<td>$\omega_{100}[\omega]$</td>
<td>$\omega_{100}[\varepsilon_0]$</td>
</tr>
<tr>
<td>$\omega_{1,1}(\omega_1)$</td>
<td>$\omega_{1,1}(\omega_1 [\omega] + 1)$</td>
<td>$\omega_{1,1}(\omega_1 [\varepsilon_0] + 1)$</td>
</tr>
<tr>
<td>$\omega_{1,3}(\omega_1)$</td>
<td>$\omega_{1,3}(\omega_1 [\omega] + 1)$</td>
<td>$\omega_{1,3}(\omega_1 [\varepsilon_0] + 1)$</td>
</tr>
<tr>
<td>$\omega_{\xi_{9}}(\omega_1)$</td>
<td>$\omega_{\xi_{9}}(\omega_1 [\omega] + 1)$</td>
<td>$\omega_{\xi_{9}}(\omega_1 [\varepsilon_0] + 1)$</td>
</tr>
<tr>
<td>$\omega_{\xi_{9}+1}(1)$</td>
<td>parameter is too big</td>
<td>parameter is too big</td>
</tr>
<tr>
<td>$\omega_{\xi_{9}+\omega_{1}+1}$</td>
<td>parameter is too big</td>
<td>parameter is too big</td>
</tr>
<tr>
<td>$\omega_{10,10}$</td>
<td>$\omega_{10,10}[\omega] + 1$</td>
<td>$\omega_{10,10}[\varepsilon_0] + 1$</td>
</tr>
<tr>
<td>$\omega_{[2,3]}\omega_4[1]$</td>
<td>parameter is too big</td>
<td>parameter is too big</td>
</tr>
<tr>
<td>$\omega_{[3]}\omega_4[1]$</td>
<td>parameter is too big</td>
<td>parameter is too big</td>
</tr>
<tr>
<td>$\omega_{[2,4]}\omega_4[1]$</td>
<td>parameter is too big</td>
<td>parameter is too big</td>
</tr>
<tr>
<td>$\omega_{[4,4]}\omega_4[1]$</td>
<td>parameter is too big</td>
<td>parameter is too big</td>
</tr>
<tr>
<td>$\omega_{[4,4]}\omega_4[1]$</td>
<td>parameter is too big</td>
<td>parameter is too big</td>
</tr>
<tr>
<td>$\omega_{[4,4]}\omega_4[1]$</td>
<td>parameter is too big</td>
<td>parameter is too big</td>
</tr>
</tbody>
</table>

Table 51: AdmisLevOrdinal::limitOrd examples.
Any integer factor, or smaller term in a parameter means it cannot be a fixed point. Section 6.3 describes pseudoCodeLevel which records this as well as the codeLevel of the term if this is not true. If that level is less than AdmisCodeLevel, false is returned.

If none of the previous tests fail, an AdmisLevOrdinal is constructed from all the parameters except that selected by the index. If this value is less than the selected parameter, true is returned and otherwise false.

9.8 Operators

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

The δ parameter in expressions 10 ([[δ]]ωκ[η]), 11 ([[δ]]ωκ,γ(β1, β2, ..., βm)) and 12 ([[δ]]ωκ[[η]]) allows a restricted version of the notation system to be embedded within itself. In this section we describe how to generalize and nest this embedding. The idea is to index the δ level with an ordinal notation, σ, and further expand the indexing with a list of these pairs. The first value of δ continues to limit the size of η. Values of δ that follow it must either be increasing or equal with increasing values of σ. κ, in turn, must be ≥ the last δ. The first δ indicates the level at which the η parameter for the ordinal as a whole and any of its parameters is restricted. Subsequent δs do not affect this. The additional index or σ can be any ordinal notation with this restriction. It is not otherwise limited.

Recall that the ordinal hierarchy beyond the Church-Kleene ordinal is somewhat like the Mandelbrot set. Any recursive formalization of the hierarchy has a well ordering less than the Church Kleene ordinal and thus it can be embedded within itself at many places and to any finite depth. This nesting must be managed to avoid an infinite descending chair or inconsistency.

The expressions for the expanded notations are as follows.

\[ [[[\delta_1, \sigma_1, \delta_2, \sigma_2, ..., \delta_m, \sigma_m]]\omega_\kappa[\eta]] \] (13)
\[ [[[\delta_1, \sigma_1, \delta_2, \sigma_2, ..., \delta_m, \sigma_m]]\omega_\kappa[[\eta]] \] (14)
\[ [[[\delta_1, \sigma_1, \delta_2, \sigma_2, ..., \delta_m, \sigma_m]]\omega_\kappa,\gamma(\beta_1, \beta_2, ..., \beta_n) \] (15)

There are several additional restrictions that limit the above ordinal notations.

1. If κ is a limit then no η parameter is allowed.
2. The most significant δ cannot be a limit.
3. If any other δ is a limit, then the associated σ must be 0.
4. If the σ associated with the least significant δ is a limit then no η parameter is allowed.
5. Similarly if the least significant δ is a limit then no η parameter is allowed.

10.1 Filling the Gaps

The idea is to partially fill the ultimately unfillable gaps between admissible ordinals. Additions to the ordinal hierarchy often expand at the top and indirectly fill in lower gaps. In this case the limit of the top is stable\(^{50}\) and only gaps are filled.

The rules that follow define α.limitOrd(υ) for any α that satisifies the above syntax and restrictions and any legal parameter υ in the existing system or an expansion of it. It should be possible to expand the system with an unbounded sequence of finite extensions that could completely define any notation in the system in terms of smaller ones through the recursive

\(^{50}\)The limit of the ordinals for which notations are defined in sections 8 through 11 is the limit of the sequence ζn where ζ0 = ω and ζn+1 = ωζn, which is ω, ωω, ωω(ωω), ωω(ωω(ωω)), ....
limitOrd algorithm. For recursive ordinals with notations in the system, notations for all smaller ordinals are in the system and accessible through either limitOrd or limitElement.

In defining a notation for \([\delta] \omega_\kappa \cdot \text{limitOrd}(v)\) with \(\kappa > \delta\) and \(\kappa\) a limit, the \(\delta\) prefix cannot be exceeded\(^{\text{51}}\). This and similar situations are designated by using limitOrdA in place of limitOrd (see Note 49). There is no actual routine limitOrdA. It is a placeholder for the specific algorithm needed to insure this. The LimitTypeInfo enum(s) (see Section 9.4 and Table 52) associated with each rule are listed at the end of the rule if applicable. If two rules apply to the same situation the first rule that matches is used.

1. Frequently, in computing \(\alpha \cdot \text{limitOrd}(v)\), a parameter of \(\alpha\), \(p\), must be replaced with \(p \cdot \text{limitOrd}(v)\). If the this occurs where the result might generate a fixed point, 1 is added to the limit Ordinal.

2. If \(\kappa\) is the least significant nonzero parameter and a limit and \(\kappa = \delta_m\) then
   \[
   \alpha = [[\delta_1, \sigma_1, \delta_2, \sigma_2, \ldots, \delta_{m-1}, \sigma_{m-1}, \delta_m]] \kappa_{\delta_m} \text{ and }
   \alpha \cdot \text{limitOrd}(\zeta) = [[\delta_1, \sigma_1, \delta_2, \sigma_2, \ldots, \delta_{m-1}, \sigma_{m-1}, \delta_m \cdot \text{limitOrdA}(\zeta)]] \omega_{\delta_m \cdot \text{limitOrdA}(\zeta)}.
   \]
   There must be a \(\delta_{m-1}\) because \(\delta_m\) cannot be a limit. Thus the two occurrences of \(\delta_m \cdot \text{limitOrdA}(\zeta)\) are adjusted identically to insure that \(\delta_m \cdot \text{limitOrdA}(\zeta)\) is a strictly increasing function that is always > \(\delta_{m-1}\) which must be < \(\delta_m\).
   Corresponding LimitTypeInfo leastLevelLimit.

3. If the least significant nonzero parameter, \(\zeta\), is a limit then \(\zeta\) in \(\alpha\) is replaced with \(\zeta \cdot \text{limitOrd}(v)\) in \(\alpha\). If the least significant nonzero parameter is \(\kappa\), \(\zeta \cdot \text{limitOrdA}(v)\) may need to be adjusted to be greater than the last and largest \(\delta_i\).
   Corresponding LimitTypeInfo paramLimit, functionLimit.

4. If \(\kappa\) is a successor and the least significant nonzero parameter (the \(\delta_i\)s and \(\sigma_i\)s are more significant) then \(\alpha \cdot \text{limitOrd}(\eta)\) appends \(\eta\) as a suffix to the notation for \(\alpha\). If \(\alpha\) contains a nonzero \(\delta\), the appended notation is \([[\eta]]\) and otherwise \([\eta]\).
   Corresponding LimitTypeInfo indexCKsuccUn, indexCKsuccEmbed.

5. If the least significant nonzero parameter is a successor, \(\beta_1\), and the next least significant parameter is \(\kappa\) a limit then the expression \(\delta < \kappa\) then the following holds.
   \[
   \alpha = [[\delta_1, \sigma_1, \delta_2, \sigma_2, \ldots, \delta_m, \sigma_m]] \omega_{\kappa(\beta_1)}.
   \]
   \[
   \alpha \cdot \text{limitOrd}(\zeta) = [[\delta_1, \sigma_1, \delta_2, \sigma_2, \ldots, \delta_m, \sigma_m]] \omega_{\kappa \cdot \text{limitOrdA}(\zeta)[[\delta_1, \delta_2, \sigma_2, \ldots, \delta_m, \sigma_m]] \omega_{\kappa(\beta_1-1)}.
   \]
   Corresponding LimitTypeInfo indexCKlimitParamUn.

6. If the least significant parameter is \(\kappa\), a limit, and and there is at least one \(\delta\) prefix then the following holds.
   \[
   \alpha = [[\delta_1, \sigma_1, \delta_2, \sigma_2, \ldots, \delta_m, \sigma_m]] \omega_{\kappa}.
   \]
   \[
   \alpha \cdot \text{limitOrd}(\zeta) = [[\delta_1, \sigma_1, \delta_2, \sigma_2, \ldots, \delta_m, \sigma_m]] \omega_{\kappa \cdot \text{limitOrdA}(\zeta)}.
   \]
   Corresponding LimitTypeInfo indexCKlimitEmbed.

7. If the least significant nonzero parameter is \(\beta_1\), a successor, the next least significant parameter is \(\kappa\), a limit and there is at least one \(\delta\) prefix then the following holds.

\(^{\text{51}}\)One could also adjust the value of \(\delta\) in this sequence.
α = [δ₁σ₁, δ₂σ₂, ..., δₘσₘ]ω₉(β₁).

\[ \text{α.limitOrd}(\zeta) = [\delta_1, \delta_2, \ldots, \delta_m, \sigma_m] \omega_{κ, \text{limitOrdA}(\zeta)}[\delta_1, \delta_2, \ldots, \delta_m, \sigma_m] \omega_{κ(\zeta - 1)}. \]

Corresponding LimitTypeInfo indexCKLimitParamEmbed

8. If \( \alpha = [\delta_1, \delta_2, \ldots, \delta_m, \sigma_m] \omega_{δ_m} \) and \( \sigma_m \) is a limit then
\[ \text{α.limitOrd}(\zeta) = [\delta_1, \delta_2, \ldots, \delta_m, \sigma_m, \text{limitOrd}(\zeta)] \omega_{δ_m}. \]
Corresponding LimitTypeInfo leastIndexLimit

9. If \( \alpha = [\delta_1, \delta_2, \ldots, \delta_m, \sigma_m] \omega_{δ_m}(\beta_1) \) with \( \beta_1 \) a successor and \( \sigma_m \) a limit then
\[ \gamma = [\delta_1, \delta_2, \ldots, \delta_m, \sigma_m] \omega_{δ_m}(\beta_1 - 1) \] and
\[ \text{α.limitOrd}(\zeta) = [\delta_1, \delta_2, \ldots, \delta_m, \sigma_m, \text{limitOrd}(\zeta)] \omega_{δ_m, b}. \]
Corresponding LimitTypeInfo leastIndexLimitParam

10. If the the least significant parameter is \( \gamma \), a successor, in for example \( \alpha = \omega_{κ, \gamma} \), then only integer values for the parameter of \( \text{limitOrd} \) are legal.
\[ \omega_{κ, \gamma} \text{limitOrd}(n) = \omega_{κ, γ-1}(\omega_{κ, γ-1} + 1, 0, ..., 0) \] where there are \( n - 1 \) zeros. If there are other more significant parameters they are copied unchanged.
Corresponding LimitTypeInfo functionSucc

11. If the \([η]\) suffix (always the lest significant if present) is a successor > 1 then only integer values for the parameter of \( \text{limitOrd} \) are legal. If \( κ > 1 \) then the following holds.
\[ \alpha.\text{limitOrd}(1) = \omega_κ[η - 1] \] and
\[ \alpha.\text{limitOrd}(n + 1) = \omega_κ-1, α.\text{limitOrd}(n+1). \]
\( δ \) and \( σ \) parameters may also be present and are copied without change if present.
Corresponding LimitTypeInfo drillDownSucc

12. If the \([η]\) suffix (always the lest significant if present) is a successor > 1 then only integer values for the parameter of \( \text{limitOrd} \) are legal. If \( κ = 1 \) then the following holds.
\[ \alpha.\text{limitOrd}(1) = \omega_κ[η - 1] \] and
\[ \alpha.\text{limitOrd}(n + 1) = ϕ.α.\text{limitOrd}(n+1). \]
Corresponding LimitTypeInfo drillDownSuccCKOne

13. If the \([η]\) suffix (if present the lest significant) is a successor > 1 then only integer values for the parameter of \( \text{limitOrd} \) are legal. For
\[ α = [\delta_1, \delta_2, \ldots, \delta_m, \sigma_m] \omega_κ[η], \]
\[ \alpha.\text{limitOrd}(1) = \omega_κ[η - 1] \] and
\[ α.\text{limitOrd}(n + 1) = \omega_κ[α.\text{limitOrd}(n)]. \]
Corresponding LimitTypeInfo drillDownSuccEmbed

14. If the suffix is \([1]\) then only integer values for the parameter of \( \text{limitOrd} \) are legal. For \( α = [\delta_1, \delta_2, \ldots, \delta_m, \sigma_m] \omega_κ[1], \alpha.\text{limitOrd}(1) = \omega \) and
\[ α.\text{limitOrd}(n + 1) = [\delta_1, \delta_2, \ldots, \delta_m, \sigma_m] \omega_κ[α.\text{limitOrd}(n)]. \]
The \( δ \) and \( σ \) prefix values are copied unchanged unless there a single \( δ_1 \) (with no sigma) and \( δ_0.κ \). In this case \( δ \) is deleted in the \( \text{limitOrd} \) results.
Corresponding LimitTypeInfo drillDownOneEmbed
15. If $\kappa$ is a successor and the least significant nonzero parameter and the next least significant parameter is $\sigma_{n}$, a limit, then $\alpha = [\delta_{1}\sigma_{1}, \delta_{2}\sigma_{2}, ..., \delta_{m}\sigma_{m}]\omega_{\kappa}$ and $\alpha.\text{limitOrd}(\zeta) = [[\delta_{1}\sigma_{1}, \delta_{2}\sigma_{2}, ..., \delta_{m-1}\sigma_{m-1}, \delta_{m}\sigma_{n}.\text{limitOrd}(\zeta)]]\omega_{\kappa}$. Corresponding LimitTypeInfo leastIndexLimit.

16. If $\kappa$, a limit, is the least significant parameter and there is no $\delta \sigma$ prefix then the following hold.

$\alpha = \omega_{\kappa}$, $\alpha.\text{limitOrd}(\zeta) = \omega_{\kappa.\text{limitOrd}(\zeta)}$.

Corresponding LimitTypeInfo indexCKlimit.

17. If $\kappa = 1$ with the suffix [1], no $\delta$ are possible. If $\alpha = \omega_{1}[1]$ and $\alpha.\text{limitOrd}(1) = \omega$ and $\alpha.\text{limitOrd}(n + 1) = \varphi_{\alpha.\text{limitOrd}(m)}$.

Corresponding LimitTypeInfo drillDownCKOne.

18. If the $\eta$ suffix is [1] then it is the least significant parameter and only integer values for the parameter of $\text{limitOrd}$ are legal.

Corresponding LimitTypeInfo drillDownOne.

(a) If $\kappa$ a successor $> 1$ and, if present, the least significant $\delta_{i} < \kappa$ then

$\alpha.\text{limitOrd}(1) = \omega_{\kappa-1}$ and $\alpha.\text{limitOrd}(n + 1) = \omega_{\kappa.\alpha.\text{limitOrd}(n)}$.

(b) If $\alpha = [\delta_{1}\sigma_{1}, \delta_{2}\sigma_{2}, ..., \delta_{m-1}\sigma_{m-1}, \delta_{m}]\omega_{\delta_{m}}$ and $\delta_{m}$ is a successor then

$\alpha.\text{limitOrd}(1) = [[\delta_{1}\sigma_{1}, \delta_{2}\sigma_{2}, ..., \delta_{m-1}\sigma_{m-1}, \delta_{m} - 1, ((\delta_{m} - 1) == \delta_{m}?1:0)]\omega_{\delta_{m}}$ and

$\alpha.\text{limitOrd}(n + 1) = [[\delta_{1}\sigma_{1}, \delta_{2}\sigma_{2}, ..., \delta_{m-1}\sigma_{m-1}, \delta_{m}.\alpha.\text{limitOrd}(n)]\omega_{\delta_{m}}$.

19. If the $[\eta]$ or $[[\eta]]$ suffix represents a limit ordinal in $\alpha$ then in $\alpha.\text{limitOrd}(\zeta) \eta$ is replaced with $\eta.\text{limitOrd}(\zeta)$.

Corresponding LimitTypeInfo drillDownLimit.

20. If the least significant nonzero parameter is $\beta_{1}$ and the next least significant parameter is $\kappa$ then only integer values for the parameter of $\text{limitOrd}$ are legal. If in addition either there is no $\delta \sigma$ prefix or the least significant $\delta$ in the prefix is $> \kappa$ then the following holds. If $\alpha = [\delta_{1}\sigma_{1}, \delta_{2}\sigma_{2}, ..., \delta_{m}\sigma_{m}]\omega_{\kappa}(\beta_{1})$ then

$\alpha.\text{limitOrd}(1) = [[\delta_{1}\sigma_{1}, \delta_{2}\sigma_{2}, ..., \delta_{m}\sigma_{m}]\omega_{\kappa}(\beta_{1}) - 1]$ and

$\alpha.\text{limitOrd}(n + 1) = [[\delta_{1}\sigma_{1}, \delta_{2}\sigma_{2}, ..., \delta_{m}\sigma_{m}]\omega_{\kappa-1.\alpha.\text{limitOrd}(n)}$.

Corresponding LimitTypeInfo indexCKsuccParam.

21. If the least significant nonzero parameter is $\beta_{1}$ and the next least significant parameter is $\kappa$ then only integer values for the parameter of $\text{limitOrd}$ are legal. If there is a single $\delta$ prefix with no $\sigma$ and $\delta = \kappa$ then the following holds. $\alpha = [[\delta]]\omega_{\delta}(\beta_{1})$. newline

$\alpha.\text{limitOrd}(1) = \omega_{\delta}(\beta_{1}) - 1$ and

$\alpha.\text{limitOrd}(n + 1) = [[\delta - 1]]\omega_{\delta-1.\alpha.\text{limitOrd}(n)+1}$.

Corresponding LimitTypeInfo indexCKsuccParamEq.

22. If the least significant nonzero parameter is $\beta_{1}$, a successor and the next least significant parameter is $\kappa$, also a successor, then only integer values for the parameter of $\text{limitOrd}$ are legal. If in addition $\delta_{m} = \kappa$ then one of the following holds.
(a) If the least significant $\delta$ has a corresponding $\sigma$ (which must be a successor) and the next smallest legal prefix deletes the least significant $\sigma, \delta$ pair then
\[
\alpha = [\delta_1, \sigma_1, \delta_2, \sigma_2, \ldots, \delta_m, \sigma_m] \omega_{\delta_m}(\beta_1).
\]
\[
\alpha.\text{limitOrd}(1) = [\delta_1, \sigma_1, \delta_2, \sigma_2, \ldots, \delta_m, \sigma_m] \omega_{\delta_m}(\beta_1 - 1).
\]
\[
\alpha.\text{limitOrd}(n + 1) = [\delta_1, \sigma_1, \delta_2, \sigma_2, \ldots, \delta_m, \sigma_m, -1, \alpha.\text{limitOrd}(n)] \omega_{\alpha.\text{limitOrd}(n)}.
\]

(b) If the least significant $\delta$ is a successor, it has a corresponding $\sigma$, the next least significant parameter is $\beta_1$ and the next smallest prefix comes from decrementing the least significant $\sigma$, then
\[
\alpha = [\delta_1, \sigma_1, \delta_2, \sigma_2, \ldots, \delta_m, \sigma_m] \omega_{\delta_m}(\beta_1).
\]
\[
\alpha.\text{limitOrd}(1) = \alpha = [\delta_1, \sigma_1, \delta_2, \sigma_2, \ldots, \delta_m, \sigma_m] \omega_{\delta_m}(\beta_1 - 1)
\]
\[
\alpha.\text{limitOrd}(n + 1) = [\alpha = [\delta_1, \sigma_1, \delta_2, \sigma_2, \ldots, \delta_m, \sigma_m - 1, \delta_m, \sigma_m, -1, \alpha.\text{limitOrd}(n)] \omega_{\alpha.\text{limitOrd}(n)}.
\]

(c) If the least significant $\delta$ has no corresponding $\sigma$ then
\[
\alpha = [\delta_1, \sigma_1, \delta_2, \sigma_2, \ldots, \delta_m, \sigma_m] \omega_{\delta_m}(\beta_1).
\]
\[
\alpha.\text{limitOrd}(1) = [\delta_1, \sigma_1, \delta_2, \sigma_2, \ldots, \delta_m, \sigma_m, -1, \delta_m] \omega_{\delta_m}(\beta_1 - 1).
\]
\[
\alpha.\text{limitOrd}(n + 1) = [\delta_1, \sigma_1, \delta_2, \sigma_2, \ldots, \delta_m, \sigma_m, -1, \delta_m, \alpha.\text{limitOrd}(n)] \omega_\kappa.
\]

23. If the two least significant nonzero parameters are successors or there is at least one zero $\beta_i$ that is less significant than the least nonzero parameter which is also a $\beta_i$, then only integer values for the parameter of limitOrd are legal. Then $\alpha.\text{limitOrd}(1) =$ the original Ordinal with the least significant nonzero parameter decremented by 1. $\alpha.\text{limitOrd}(n + 1) =$ the original Ordinal when the more significant parameters decremented by 1 and $\alpha.\text{limitOrd}(n)$ substituted in the next least significant parameter whether or not it was zero).

24. If the least significant nonzero parameter is a successor and the next least nonzero parameter, $\zeta$, is a limit then $\alpha.\text{limitOrd}(\nu) =$ the original Ordinal with $\zeta$ replaced by $\zeta.\text{limitOrd}(\nu)$ and the next least significant parameter replaced by the original Ordinal with least significant nonzero parameter decremented by 1.

25. If the least significant nonzero parameter is $\beta$ and $\kappa$ is a limit equal to $\delta_m$ and the next least significant parameter then the following holds.
\[
\alpha = [\delta_1, \sigma_1, \delta_2, \sigma_2, \ldots, \delta_m, \sigma_m] \omega_{\delta_m}(\beta_1).
\]
\[
\alpha.\text{limitOrd}(\zeta) = [\delta_1, \sigma_1, \delta_2, \sigma_2, \ldots, \delta_m, \sigma_m, -1, \delta_m, \alpha.\text{limitOrd}(\zeta)] \omega_{\delta_m}(\alpha.\text{limitOrd}(\zeta), [\delta_1, \sigma_1, \delta_2, \sigma_2, \ldots, \delta_m, \sigma_m] \omega_{\delta_m}(\beta_1 - 1) + 1.
\]

Following are two tables based on the enum, $\text{LimitTypeInfo}$. The first gives conditions for each case, the associated $\text{limitType}$ and the number of the above rule that applies (including a hyperlink). The first table links to the second through entries in the $\text{LimitTypeInfo}$
column. This second table gives an example notation, $\alpha$, and information about it for each entry. This includes the exit code of the `limitElement` routine that computed the result with a hyperlink to the table entry documenting that routine. The entry also includes the two computed values $\alpha.limitOrd(1)$ and $\alpha.limitOrd(2)$. Note for finite parameters `limitOrd` and `limitElement` are equivalent.
\[
\begin{align*}
\beta_i & \text{ defined in: } 7, 8, 11, 15, \\
\gamma & \text{ defined in: } 7, 8, 11, 15, \\
\eta & \text{ defined in: } 9, 10, 12, 13, 14, \\
\kappa & \text{ defined in: } 8, 9, 10, 11, 12, 13, 15, 14, \\
\delta & \text{ defined in: } 10, 11, 12, 13, 15, 14, \\
\sigma & \text{ defined in: } 13, 15, 14.
\end{align*}
\]

<table>
<thead>
<tr>
<th>LimitTypeInfo</th>
<th>Least significant nonzero parameters</th>
<th>limitType</th>
<th>R</th>
</tr>
</thead>
<tbody>
<tr>
<td>unknownLimit</td>
<td>used internally</td>
<td></td>
<td></td>
</tr>
<tr>
<td>zeroLimit</td>
<td>the ordinal zero</td>
<td>nullLimitType</td>
<td></td>
</tr>
<tr>
<td>finiteLimit</td>
<td>a successor ordinals</td>
<td>nullLimitType</td>
<td></td>
</tr>
<tr>
<td>paramSucc</td>
<td>in (\omega^x x) is a successor</td>
<td>integerLimitType</td>
<td></td>
</tr>
<tr>
<td>paramLimit</td>
<td>(\beta_i) is a limit</td>
<td>(\beta_i).limitType</td>
<td>3</td>
</tr>
<tr>
<td>paramSuccZero</td>
<td>successor (\beta_i) followed by at least one zero</td>
<td>integerLimitType</td>
<td>25</td>
</tr>
<tr>
<td>paramsSucc</td>
<td>least significant (\beta_i) is successor</td>
<td>integerLimitType</td>
<td>23</td>
</tr>
<tr>
<td>paramNxtLimit</td>
<td>Limit (\beta_i), zero or more zeros and successor.</td>
<td>(\beta_i).limitType()</td>
<td>24</td>
</tr>
<tr>
<td>functionSucc</td>
<td>(\gamma) is successor and (\beta_i == 0).</td>
<td>integerLimitType</td>
<td>10</td>
</tr>
<tr>
<td>functionNxtSucc</td>
<td>(\gamma) and a single (\beta_i) are successors.</td>
<td>integerLimitType</td>
<td>23</td>
</tr>
<tr>
<td>functionLimit</td>
<td>(\gamma) is a limit and (\beta_i == 0).</td>
<td>(\gamma).limitType</td>
<td>3</td>
</tr>
<tr>
<td>functionNxtLimit</td>
<td>(\gamma) is a limit and a single (\beta_i) is a successor.</td>
<td>(\gamma).limitType</td>
<td>24</td>
</tr>
<tr>
<td>drillDownLimit</td>
<td>(\eta) or ([\eta]) suffix is a limit.</td>
<td>(\eta).limitType</td>
<td>19</td>
</tr>
<tr>
<td>drillDownSuccCKOne</td>
<td>(\eta) is a successor (\kappa == 1).</td>
<td>integerLimitType</td>
<td>11</td>
</tr>
<tr>
<td>drillDownSuccEmbed</td>
<td>(\eta) is a successor (\kappa &gt; 1).</td>
<td>integerLimitType</td>
<td>12</td>
</tr>
<tr>
<td>drillDownOne</td>
<td>((\eta == 1)) &amp; ((\kappa &gt; 1)) &amp; (\eta) is a successor.</td>
<td>integerLimitType</td>
<td>18b</td>
</tr>
<tr>
<td>drillDownOneEmbed</td>
<td>([\eta]) is a successor (\kappa &gt; 1).</td>
<td>integerLimitType</td>
<td>14</td>
</tr>
<tr>
<td>drillDownCKOne</td>
<td>(\eta == \kappa == 1).</td>
<td>integerLimitType</td>
<td>17</td>
</tr>
<tr>
<td>indexCKlimit</td>
<td>(\kappa) is a limit and (\delta == 0).</td>
<td>(\kappa).limitType</td>
<td>16</td>
</tr>
<tr>
<td>indexCKsuccParam</td>
<td>(\kappa) and a single (\beta) are successors &amp; (\kappa \neq \delta)</td>
<td>integerLimitType</td>
<td>20</td>
</tr>
<tr>
<td>indexCKsuccParamEq</td>
<td>(\kappa) and a single (\beta) are successors &amp; (\kappa == \delta)</td>
<td>integerLimitType</td>
<td>21</td>
</tr>
<tr>
<td>indexCKsuccUn</td>
<td>(\kappa) is a successor and (\delta &gt; 0)</td>
<td>indexCKtoLimitType</td>
<td>4</td>
</tr>
<tr>
<td>indexCKlimiteEmbed</td>
<td>(\kappa) is a successor and (\delta == 0)</td>
<td>indexCKtoLimitType</td>
<td>4</td>
</tr>
<tr>
<td>indexCKlimiteEmbed</td>
<td>(\kappa) is a limit and (\beta_1) is a successor.</td>
<td>(\kappa).limitType</td>
<td>5</td>
</tr>
<tr>
<td>indexCKlimiteEmbed</td>
<td>(\kappa) is limit and (\delta &gt; 0)</td>
<td>(\kappa).limitType</td>
<td>6</td>
</tr>
<tr>
<td>indexCKlimiteParamEq</td>
<td>(\kappa) a limit (\kappa \neq \delta)</td>
<td>(\kappa).limitType</td>
<td>7</td>
</tr>
<tr>
<td>leastIndexLimit</td>
<td>least significant (\sigma_i) is a limit</td>
<td>(\sigma_i).limitType</td>
<td>8</td>
</tr>
<tr>
<td>leastLevelLimit</td>
<td>least significant (\delta_i) is a limit</td>
<td>(\delta_i).limitType</td>
<td>2</td>
</tr>
<tr>
<td>leastIndexSuccParam</td>
<td>(\beta_1) and least significant (\sigma_i) are successors</td>
<td>integerLimitType</td>
<td>22a</td>
</tr>
<tr>
<td>leastIndexSuccParam</td>
<td>(\beta_1) and least significant (\delta_i) are successors</td>
<td>integerLimitType</td>
<td>22c</td>
</tr>
<tr>
<td>leastIndexLimitParam</td>
<td>least significant (\sigma_i) is a limit, (\beta_1) successor</td>
<td>(\sigma_i).limitType</td>
<td>9</td>
</tr>
<tr>
<td>leastLevelLimitParam</td>
<td>least (\delta_1 == \kappa) both limits, (\beta_1) successor</td>
<td>(\delta_1).limitType</td>
<td>25</td>
</tr>
</tbody>
</table>

Table 52: LimitTypeInfo descriptions through nestedEmbedCodeLevel
\[ \begin{array}{|c|c|c|}
\hline
\text{LimitTypeInfo} & \text{Ordinal} & \text{Limit} \\
\hline
\text{LimitElement} & 1 & \omega^{+12} \\
\hline
\end{array} \]

The following is at code level \texttt{cantorCodeLevel} (Section 4) and above.

\[
\begin{array}{l}
\text{paramSucc} = \omega^{+11} \\
\text{paramLimit} = \varphi(\omega, 0) \\
\text{paramSuccZero} = \varphi(\omega + 12, 0, 0) \\
\text{paramSucc} = \varphi(\omega + 2, 5, 0) \\
\text{paramNxtLimit} = \varphi(\omega, \varphi(\omega, 0, 0, 32, 1, 0, 33, 0, \omega^{+12})) \\
\end{array}
\]

The following is at code level \texttt{finiteFuncCodeLevel} (Section 6) and above.

\[
\begin{array}{l}
\text{functionSucc} = \varphi_{11} \\
\text{functionNxtSucc} = \varphi_{14}(4) \\
\text{functionLimit} = \varphi_{1}(\varphi(3, 1, 0) + 1) \\
\text{functionNxtLimit} = \varphi_{1}(\varphi(3, 14, 0) + 1) \\
\end{array}
\]

The following is at code level \texttt{iterFuncCodeLevel} (Section 7) and above.

\[
\begin{array}{l}
\text{drillDownLimit} = [2, \omega^{+12}] \\
\text{drillDownSucc} = [2, \omega^{+12}] \\
\text{drillDownSuccCKOne} = [2, \omega^{+12}] \\
\text{drillDownSuccEmbed} = [3, \omega^{+12}] \\
\text{drillDownOne} = [3, \omega^{+12}] \\
\text{drillDownOneEmbed} = [3, \omega^{+12}] \\
\text{drillDownCKOne} = [3, \omega^{+12}] \\
\text{indexCKLimit} = [3, \omega^{+12}] \\
\text{indexCKSuccParam} = [3, \omega^{+12}] \\
\text{indexCKSuccParamEq} = [3, \omega^{+12}] \\
\text{indexCKSuccUn} = [2, \omega^{+12}] \\
\text{indexCKLimitParamUn} = [3, \omega^{+12}] \\
\text{indexCKLimitEmbed} = [3, \omega^{+12}] \\
\end{array}
\]

The following is at code level \texttt{nestedEmbedCodeLevel} (Section 10) and above.

\[
\begin{array}{l}
\text{indexCKLimitParamEmbed} = [4, \omega^{+12}] \\
\text{leastLimit} = [10, \omega^{+12}] \\
\text{leastLevelLimit} = [10, \omega^{+12}] \\
\text{leastLevelSuccParam} = [12, \omega^{+12}] \\
\text{leastLevelSuccUn} = [3, \omega^{+12}] \\
\text{leastIndexLimitParam} = [34, \omega^{+12}] \\
\text{leastLevelLimitParam} = [3, \omega^{+12}] \\
\end{array}
\]

Table 53: \texttt{limitElement} and \texttt{LimitTypeInfo} examples through \texttt{nestedEmbedCodeLevel}

87
11 NestedEmbedOrdinal class

Class `NestedEmbedOrdinal` partially fills the gaps in the countable admissible hierarchy as defined by the `AdmisLevOrdinal` class in Section 9 and discussed in Section 10.1. Class `NestedEmbedOrdinal` is derived from `AdmisLevOrdinal` and its base classes. As always the ordinal notations it defines can be used in all the base classes it is derived from down to `Ordinal`.

C++ class `NestedEmbedOrdinal` uses the notation in expressions [13 through 15]. In the ordinal calculator the ‘.’ character is replaced with ‘/’. Examples of this are shown in Table 54. The parameter to the left of ‘.’ is referred to as the level and the one to the right as the index. The number of entries in the prefix in double square brackets is the number of \( \delta_i \sigma_i \) pairs. The level or \( \delta_i \) cannot be 0. The index (and \( \sigma_i \)) are omitted if \( \sigma_i = 0 \).

The class for a single term of a `NestedEmbedOrdinal` is `NestedEmbedNormalElement`. It is derived from `AdmisNormalElement` and its base classes.

Only the prefix in double square brackets is new. All other `NestedEmbedOrdinal` parameters are the same as those in `AdmisLevOrdinals`. The prefix parameters must either include two \( \delta_i \sigma_i \) entries with a non zero value for level or one entry with a nonzero value for both the level and index. If this condition is not met than the notation defines an `AdmisLevOrdinal`.

The `NestedEmbedNormalElement` class should not be used directly to create ordinal notations. Instead use function `nestedEmbedFunctional`. Often the easiest way to define an ordinal at this level is interactively in the ordinal calculator. The command `cppList` can generate C++ code for any ordinals defined interactively. Some examples are shown in Figure 2. This figure is largely self explanatory except for the classes `IndexedLevel` and `NestedEmbeddings` used to define the double bracketed prefix. `IndexedLevel` is a \( \delta_i \sigma_i \) pair. `NestedEmbeddings` is a class containing an array of pointers to `IndexedLevels` and a flag that indicates if the `drillDown` suffix has double square brackets.

11.1 compare member function

`NestedEmbedNormalElement::compare` is called from a notation for a single term in the form of expressions [13 through 15]. It compares the `CantorNormalElement` parameter, `trm`, against the `NestedEmbedNormalElement` instance `compare` is called from. As with `AdmisLevOrdinals` and its base classes, the work of comparing `Ordinals` with multiple terms is left to the `Ordinal` and `OrdinalImpl` base class functions which call the virtual functions that operate on a single term.

`NestedEmbedNormalElement::compare`, with a `CantorNormalElement` and an ignore factor flag as arguments, overrides `AdmisNormalElement::compare` with the same arguments (see Section 9.1). It outputs 1, 0 or -1 if the object it is called from is greater than equal to or less than its argument. There are two versions of `NestedEmbedNormalElement::compare` the first has two arguments as described above. The second is for internal use only and has two additional context arguments, the context for the base function and the context for the argument. The context is expanded from that in Section 9.1 using an expanded base class `NestedEmbeddings` from `Embeddings`. The expanded version includes the entire double square bracketed prefix in force at this stage of the compare.

`compare` first checks if its argument’s `codeLevel` is > `nestedEmbedCodeLevel`. If so
//a = [[w + 1, w + 1/1]]omega_{ w + 1}[ 1]
static const IndexedLevel * const a0IndexedLevel[] = {
    new IndexedLevel(( * new Ordinal(expFunctional(Ordinal::one).getImpl()
        .addLoc(Ordinal::one))) ,Ordinal::zero),
    new IndexedLevel(( * new Ordinal(expFunctional(Ordinal::one).getImpl()
        .addLoc(Ordinal::one))) ,Ordinal::one),
    NULL
};
const Ordinal& a = nestedEmbedFunctional(
    ( * new Ordinal(expFunctional(Ordinal::one).getImpl()
        .addLoc(Ordinal::one))),
    Ordinal::zero,
    (* new NestedEmbeddings( a0IndexedLevel, false)),
    NULL,
    Ordinal::one
);

//b = [[1, 1/1]]omega_{ 1}[ 1]
static const IndexedLevel * const b1IndexedLevel[] = {
    new IndexedLevel(Ordinal::one ,Ordinal::zero),
    new IndexedLevel(Ordinal::one ,Ordinal::one),
    NULL
};
const Ordinal& b = nestedEmbedFunctional(
    Ordinal::one,
    Ordinal::zero,
    (* new NestedEmbeddings( b1IndexedLevel, false)),
    NULL,
    Ordinal::one
);

Figure 2: Defining NestedEmbedOrdinals with cppList
it calls a higher level routine by calling its argument’s member function. The code level of the class object that `NestedEmbedNormalElement::compare` is called from should be `nestedEmbedCodeLevel`.

The prefix list of $\delta_{i} \omega_{i}$ pairs (from expressions 13 through 15) makes comparisons context sensitive just as the $\delta$ prefix does for an `AdmisNormalElement`. This context is passed as the `nestedEmbeddings` member variable of a `NestedEmbedNormalElement`. The same structure can also be reference as `embeddings` in the `AdmisNormalElement` base class. The context is relevant to comparisons of internal parameters of two notations being compare. Thus the context sensitive version of compare passes `embeddings` to compares that are called recursively.

The context sensitive version of the virtual function `compare` has four arguments.

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

Table 54: `NestedEmbedOrdinal` interpreter code examples

<table>
<thead>
<tr>
<th>Interpreter code</th>
<th>Ordinal notation</th>
</tr>
</thead>
<tbody>
<tr>
<td>[[1, 2]]w_{ 2}</td>
<td>[1, 2]</td>
</tr>
<tr>
<td>[[1/1]]w_{ 1}</td>
<td>[1]</td>
</tr>
<tr>
<td>[[1/1]]w_{ 1}[ 5]</td>
<td>[1]</td>
</tr>
<tr>
<td>[[1/1]]w_{ 1}[[ 12]]</td>
<td>[1]</td>
</tr>
<tr>
<td>[[1, 1/1]]w_{ 1}[ 1]</td>
<td>[1]</td>
</tr>
<tr>
<td>[[1, w]]w_{ w}</td>
<td>[1,w]</td>
</tr>
<tr>
<td>[[1, 2/1]]w_{ 2}</td>
<td>[1,2]</td>
</tr>
<tr>
<td>[[1, 2/w]]w_{ w}</td>
<td>[1,2]</td>
</tr>
<tr>
<td>[[1, 2/w]]w_{ w}</td>
<td>[1,2]</td>
</tr>
<tr>
<td>[[3, 12]]w_{ w}</td>
<td>[3,12]</td>
</tr>
<tr>
<td>[[1/1, 1/2]]w_{ 1}</td>
<td>[1]</td>
</tr>
<tr>
<td>[[1, 2]]w_{ 2}[ 1]</td>
<td>[1,2]</td>
</tr>
<tr>
<td>[[1, 2]]w_{ 2}[[ 8]]</td>
<td>[1,2]</td>
</tr>
<tr>
<td>[[1, 1/1]]w_{ 1}[ 31]</td>
<td>[1,1,1]</td>
</tr>
<tr>
<td>[[2, 2/1]]w_{ 20}[ 1]</td>
<td>[2,2,1]</td>
</tr>
<tr>
<td>[[1, 2, 3, 4/1]]w_{ 4}[ 1]</td>
<td>[1,2,3,4]</td>
</tr>
<tr>
<td>[[w + 1/1]]w_{ w + 1}[ 1]</td>
<td>[w + 1,1]</td>
</tr>
<tr>
<td>[[1, 2, 2/5, w + 1]]w_{ w + 12}[ 1]</td>
<td>[1,2,2,5, ω + 1]</td>
</tr>
<tr>
<td>[[w + 1, w + 1/1]]w_{ w + 1}[ 1]</td>
<td>[w + 1,ω + 1,1]</td>
</tr>
</tbody>
</table>

90
As with `compare` for `AdmisLevOrdinals` and its base `classes`, this function depends on the `getMaxParameter()` that returns the maximum value of all the parameters (except $\eta$ and the list of $\delta \sigma$ pairs) in expressions 13 through 15.

The logic of `NestedEmbedNormalElement::compare` with the above four parameters is summarized in Table 56. This is an extension of Table 32 and uses definitions from that table and depends on the `NestedEmbeddings::compare` function.

### 11.2 class NestedEmbeddings

`NestedEmbeddings` is defined within class `NestedEmbedNormalElement` using the base class `Embeddings`. It contains the list of $\delta_i \sigma_i$ pairs and a flag `Embeddings::isDrillDownEmbed` indicating if the prefix is $[\eta]$ (returns `false`) or $[[\eta]]$ (returns `true`). It contains a variety of other flags and functions to facilitate operations on `Embeddings` and `NestedEmbeddings`. Of particular importance is the ability to `compare` the size of two `Embeddings` and to find the `nextLeast` most significant index (assuming the least significant parameter is not a limit). These latter two routines are documented next.

#### 11.2.1 `compare` member function

`compare` with a single parameter of class `Embeddings` is a virtual function of both `NestedEmbeddings` and its base `class`. It returns 1, 0 or -1 if its argument is less than, equal to or greater than the `NestedEmbeddings` instance it is called from. The comparison starts with the most significant $\delta$ in the list of $\delta \sigma$ pairs in the notation prefixes. If they are not equal the result of this comparison is returned. Otherwise the corresponding $\sigma$s are compared. Again if a there is a difference that result is returned. If the first pairs are identical the same test is applied to the second pair. This is repeated until there is no further $\delta$ or $\sigma$ to check in one or both pairs. If this true of only pair then that pair is greater than the other. If it is true of both they are equal.

When checking against an argument that is of only type `Embeddings` there is at most a single $\delta$ value to compare. If there is no $\delta$ value to compare the comparison is undefined. In that case the one with no $\delta$ value is arbitrarily considered to be greater, because this is helpful in some cases. In others cases one needs an additional check for this.

#### 11.2.2 `nextLeast` member function

This is a member function of both `classes` `Embeddings` and `NestedEmbeddings` but it is only used at the `nestedEmbedCodeLevel` and above. It computes the next smallest version of `NestedEmbeddings` or `Embeddings` consistent with the the restrictions on notations that immediately follow expressions 13 to 15. It also sets an enum `howCreated`. This is used in Table 58.

### 11.3 `limitElement` member function

`NestedEmbedNormalElement::limitElement` overrides base functions starting with `AdmisNormalElement::limitElement` (see Section 9.2). It operates on a single term of
Table 55: Symbols used in compare Table 56

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>effEbd</td>
<td>effectiveEmbedding(embed)</td>
</tr>
<tr>
<td>effectiveEmbedding</td>
<td>returns the maximum embedding of self and parameter embedding</td>
</tr>
<tr>
<td>embed</td>
<td>the $[[]]$ prefix of an ordinal notation</td>
</tr>
<tr>
<td>embIx</td>
<td>embedIndex from effEbd</td>
</tr>
<tr>
<td>nestEmbed</td>
<td>nestedEmbeddings</td>
</tr>
<tr>
<td>termEmbed</td>
<td>embedding for compare parameter trm</td>
</tr>
<tr>
<td>trmEffEbd</td>
<td>trm.effectiveEmbeddings(termEmbed)</td>
</tr>
<tr>
<td>trmIxCK</td>
<td>indexCK parameter of trm</td>
</tr>
<tr>
<td>trmPmRst</td>
<td>termParamRestrict (term has nonzero embedding)</td>
</tr>
</tbody>
</table>

See tables 31 and 55 for symbol definitions.

Comparisons use Embeddings context for both operands.

Value is $\text{ord.compare(trm)} = -1$, 0 or 1 if ord $<$, $=$ or $>$ trm

<table>
<thead>
<tr>
<th>X</th>
<th>Condition</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>NEA1</td>
<td>$(\text{diff} = \text{parameterCompare(trm)}) \neq 0$</td>
<td>diff</td>
</tr>
<tr>
<td>NEA2</td>
<td>$((\text{diff} = \text{nestEmbed.compare(trmEffEbd)}) \neq 0) \land \text{trmPmRst}$</td>
<td>diff</td>
</tr>
<tr>
<td>NEA3</td>
<td>$\text{trm.codeLevel} &lt; \text{admisCodeLevel}$</td>
<td>1</td>
</tr>
</tbody>
</table>
| NEA4| $\text{trm.codeLevel} > \text{nestedEmbedCodeLevel}$  
  $\text{diff} = -\text{trm.compare(*this)}$ | diff  |
| NECA| $(\text{diff} = (\text{trmIxCK.compare(embIx)} \geq 0)?-1:1) \neq 0$  
  $!\text{trmPmRst}$ | diff  |
| NECZ| $\text{diff} = \text{AdmisNormalElement::compareCom(trm)}$ (see Table 33)  
  if none of the above conditions are met | diff  |

Table 56: NestedEmbedNormalElement::compare summary
the normal form expansion and does the bulk of the work. It takes a single integer parameter. Increasing values for the argument yield larger ordinal notations as output. In this version, as in `AdmisNormalElement::limitElement`, the union of the ordinals represented by the outputs for all integer inputs are not necessarily equal to the ordinal represented by the `NestedEmbedNormalElement` class instance `limitElement` is called from. They are equal if the `limitType` of this instance of `NestedEmbedNormalElement` is `integerLimitType` (see Section 8.3).

As usual `Ordinal::limitElement` does the work of operating on all but the last term of an `Ordinal` by copying all but the last term of the result unchanged from the input `Ordinal`. The last term is generated based on the last term of the `Ordinal` instance `limitElement` is called from.

`limitElement` calls `drillDownLimitElement` or `drillDownLimitElementEq` when there is a nonzero value for the `[]` or `[][]` suffix. The “Eq” version is called if $\kappa = \delta_m$ and the `[][]` prefix needs to be modified. It calls `embedLimitElement` in other cases where it is necessary to modify the `[][]` prefix. These four routines are outlined in tables 57 to 59. Each table references a second table of examples using exit codes (column X) to connect lines in each pair of tables.

One complication is addressed by routine `AdmisNormalElement::increasingLimit`. This is required when $\kappa$ or the least significant level in the embedding is a limit and needs to be expanded. This routine selects an element from a sequence whose union is $\kappa$ while insuring that this does not lead to a value less than the effective $\delta$. This selection must be chosen so that increasing inputs produce increasing outputs and the output is always less than the input. The same algorithm is used for `limitOrd`. See Note 49 on page 74 for a description of the algorithm.
\( \alpha \) is a notation for one of expressions 13 to 15. 

\[
[[\delta_1, \sigma_1, \delta_2, \sigma_2, ..., \delta_m, \sigma_m]] \omega_\kappa[[\eta]] \\
[[\delta_1, \sigma_1, \delta_2, \sigma_2, ..., \delta_m, \sigma_m]] \omega_{\kappa, \gamma}(\beta_1, \beta_2, ..., \beta_m) \\
[[\delta_1, \sigma_1, \delta_2, \sigma_2, ..., \delta_m, \sigma_m]] \omega_\kappa[[\eta]]
\]

<table>
<thead>
<tr>
<th>( X )</th>
<th>Condition</th>
<th>LimitTypeInfo</th>
<th>( \alpha \leq (n) )</th>
</tr>
</thead>
<tbody>
<tr>
<td>NLA</td>
<td></td>
<td>paramLimit</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>paramSuccZero</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>paramsSucc</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>paramNxtLimit</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>functionSucc</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>functionNxtSucc</td>
<td>AdmisNormalElement::limitElementCom(n)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>functionNxtLimit</td>
<td>See Table 37.</td>
</tr>
<tr>
<td>NLB</td>
<td>( \delta_m = \kappa )</td>
<td>drillDownSucc</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>drillDownSuccCKOne</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>drillDownOne</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>drillDownCKOne</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>drillDownLimit</td>
<td>drillDownLimitElementEq(n)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>drillDownSuccCKOne</td>
<td>See Table 58.</td>
</tr>
<tr>
<td>NLC</td>
<td>( \delta_m \neq \kappa )</td>
<td>drillDownLimit</td>
<td>AdmisNormalElement::limitElementCom(n)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>drillDownSuccCKOne</td>
<td>See Table 39.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>drillDownOne</td>
<td>AdmisNormalElement::limitElementCom(n)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>drillDownCKOne</td>
<td>See Table 39.</td>
</tr>
<tr>
<td>NLC</td>
<td></td>
<td>drillDownSuccEmbed</td>
<td>AdmisNormalElement::limitElementCom(n)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>drillDownCKOneEmbed</td>
<td>See Table 39.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>drillDownOneEmbed</td>
<td>See Table 39.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>indexCKLimit</td>
<td>embedLimitElement(n)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>indexCKsucceEmbed</td>
<td>See Table 59.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>indexCKsucceUn</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>indexCKLimitParamUn</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>indexCKLimitEmbed</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>leastIndexLimit</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>leastIndexLimitParam</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>leastLevelLimit</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>leastIndexLimitParamEmbed</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>indexCKLimitParamEmbed</td>
<td></td>
</tr>
<tr>
<td>NLE</td>
<td></td>
<td>leastIndexSuccParam</td>
<td>paramLimitElement(n)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>leastLevelSuccParam</td>
<td>See Table 60.</td>
</tr>
<tr>
<td>NLP</td>
<td></td>
<td>leastIndexSuccParam</td>
<td>paramLimitElement(n)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>leastLevelSuccParam</td>
<td>See Table 60.</td>
</tr>
</tbody>
</table>

See Table 61 for examples.

Table 57: NestedEmbedNormalElement::limitElement cases
\[ \alpha \text{ is a notation from expression 13.} \]
\[ [[\delta_1, \sigma_1, \delta_2, \sigma_2, \ldots, \delta_m, \sigma_m]] \omega_{\kappa}[\eta] \]
This expression can also be written as: \[ [[\Delta_m]] \omega_{\kappa}[\eta] \]

The following satisfy \( \eta > 0 \land \sigma_m.isLm \land (!\sigma_m.isZ \lor !\delta_m.isLm) \).

All entries assume \( sz==0 \land (\lambda == 0) \land \eta.isSc \land !isDdEmb \).
They use the \([[[\delta_1', \sigma_1', \delta_2', \sigma_2', \ldots, \delta_m', \sigma_m']]]\) prefix (\text{se or smallerEmbed}) which is the
next smallest (nextLeast) \text{Embeddings}. See Section 11.2.2.
All but the first entry initialize \( b \) and return this value when \( n == 1 \).
if \( (\eta==1) \) \( b=[[\delta_1, \sigma_1, \delta_2, \sigma_2, \ldots, \delta_m, \sigma_m]]\omega_{\kappa} \);
else \( b=[[\delta_1, \sigma_1, \delta_2, \sigma_2, \ldots, \delta_m, \sigma_m]]\omega_{\delta_m}[\eta-1] \);
\((\text{se.el == bel})\) is smallerEmbed.embedLevel == Embeddings::baseEmbedLevel
\((\text{se.el == bel})\) if \( \text{se} \) is the prefix for an \text{AdmisLevOrdinal}.

The LimitTypeInfo options for this table are in the NLB exit code entry in Table 57.
The \text{howCreated} field is for an enum from \text{class NestedEmbeddings}. See Section 11.2.

<table>
<thead>
<tr>
<th>X</th>
<th>Condition(s)</th>
<th>howCreated</th>
<th>( \alpha.le(n) )</th>
</tr>
</thead>
<tbody>
<tr>
<td>DQA</td>
<td>(se.el == bel)</td>
<td>NA</td>
<td>if ( (\eta==1) ) ( b=[[\delta_1']][\omega_{\kappa}] );\nelse ( b=[[\delta_1, \sigma_1, \delta_2, \sigma_2]][\omega_{\delta_m}[\eta-1]] );\nfor ( (i==1;i&lt;n;i++) ) ( b=[[\delta_1']][\omega_{b}] ).</td>
</tr>
<tr>
<td>DQB</td>
<td>indexDecr2</td>
<td>indexDecr</td>
<td>( [[\delta_1', \sigma_1', \delta_2', \sigma_2', \ldots, \delta_m', \sigma_m'], b]]\omega_{b}; \text{Rtn} b ).</td>
</tr>
<tr>
<td>DQC</td>
<td>( \delta_m.isLm )</td>
<td>levelDecrIndexSet</td>
<td>for ( (i==1;i&lt;n;i++) ) ( b=[[\delta_1', \sigma_1', \delta_2', \sigma_2', \ldots, \delta_{m-1}, \sigma_{m-1}, \delta_m', \sigma_m', b]][\omega_{b}]; \text{Rtn} b ).</td>
</tr>
<tr>
<td>DQD</td>
<td></td>
<td>levelDecrIndexSet</td>
<td>for ( (i==1;i&lt;n;i++) ) ( b=[[\delta_1', \sigma_1', \delta_2', \sigma_2', \ldots, \delta_{m-1}, \sigma_{m-1}, \delta_m', b.list][\omega_{b}]; \text{Rtn} b ).</td>
</tr>
<tr>
<td>DQE</td>
<td></td>
<td>deleteLeast</td>
<td>for ( (i==1;i&lt;n;i++) ) ( b=[[\delta_1', \sigma_1', \delta_2', \sigma_2', \ldots, \delta_m', \sigma_m'][\omega_{b}]; \text{Rtn} b ).</td>
</tr>
</tbody>
</table>

See Table 62 for examples.

Table 58: NestedEmbedNormalElement::drillDownLimitElementEq cases
Symbols used here and in Table 66
See Table 35 for additional symbols.

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>le</td>
<td>limitElement (see Table 57)</td>
</tr>
<tr>
<td>loa</td>
<td>adjusted limitOrd or limitElement that takes into account the constraints on $\kappa$, $\delta$ and $\sigma$ from more significant parameters. This is done with leUse (see Note 48) and increasingLimit (see Note 49).</td>
</tr>
</tbody>
</table>

$\alpha$ is a notation for expression 13 with $(\eta == 0) \land (\gamma == 0)$:

$$[[\delta_1 \sigma_1, \delta_2 \sigma_2, \ldots, \delta_m \sigma_m]] \omega_k \lor [[\Delta_m]] \omega_k$$

See table 35 for additional symbol definitions.

The following $\alpha$ satisfies $(\eta == 0) \land (\gamma == 0) \land (sz < 2)$

<table>
<thead>
<tr>
<th>X</th>
<th>LimitTypeInfo</th>
<th>$\alpha.1e(n)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>NEA</td>
<td>indexCKsuccEmbed</td>
<td>$[[\delta_1 \sigma_1, \delta_2 \sigma_2, \ldots, \delta_m \sigma_m]] \omega_k[n]$</td>
</tr>
<tr>
<td>NEB</td>
<td>leastIndexLimitParam</td>
<td>$b = [[\delta_1 \sigma_1, \delta_2 \sigma_2, \ldots, \delta_m \sigma_m]] \omega_k(\beta_1 - 1)$; Rtn $[[\delta_1 \sigma_1, \ldots, \delta_m \sigma_m, \loa(n)]] \omega_{\delta_m, \loa(n), b}$</td>
</tr>
<tr>
<td>NEC</td>
<td>leastLevelLimitParam</td>
<td>$b = [[\delta_1 \sigma_1, \delta_2 \sigma_2, \ldots, \delta_m \sigma_m]] \omega_k(\beta_1 - 1)$; Rtn $[[\delta_1 \sigma_1, \ldots, \delta_m \sigma_m, \loa(n)]] \omega_{\delta_m, \loa(n), b}$</td>
</tr>
</tbody>
</table>
| NED    | indexCKlimitEmbed | $[[\delta_1 \sigma_1, \delta_2 \sigma_2, \ldots, \delta_m \sigma_m]] \omega_k(\beta_1 - 1)$; Rtn $[[\delta_1 \sigma_1, \delta_2 \sigma_2, \ldots, \delta_m \sigma_m, \loa(n)]] \omega_k 
| NEF    | leastIndexLimit | $[[\delta_1 \sigma_1, \delta_2 \sigma_2, \ldots, \delta_m \sigma_m, \loa(n)]] \omega_k$ |
| NEG    | leastLevelLimit | $[[\delta_1 \sigma_1, \ldots, \delta_m \sigma_m, \loa(n)]] \omega_{\delta_m, \loa(n)}$ |
| NEH    | indexCKlimitParamEmbed | $b = [[\delta_1 \sigma_1, \delta_2 \sigma_2, \ldots, \delta_m \sigma_m]] \omega_k(\beta_1 - 1)$; Rtn $[[\delta_1 \sigma_1, \delta_2 \sigma_2, \ldots, \delta_m \sigma_m]] \omega_k, \loa(n), b, 1p1$ |

See Table 63 for examples.

Table 59: NestedEmbedNormalElement::embedLimitElement cases
\[\alpha \text{ is a notation for one of expressions 13 to 15.}\]

\[
\begin{align*}
\lbrack [\delta_1, \sigma_1, \delta_2, \sigma_2, \ldots, \delta_m, \sigma_m] \omega_{\kappa, \gamma} & (\beta_1, \beta_2, \ldots, \beta_m) \\
\lbrack [\delta_1, \sigma_1, \delta_2, \sigma_2, \ldots, \delta_m, \sigma_m] \omega_{\kappa} & (\eta) \\
\end{align*}
\]

All entries assume \(sz == 1 \wedge (\lambda == 0)\).

They initialize \(b = \lbrack [\delta_1, \sigma_1, \delta_2, \sigma_2, \ldots, \delta_m, \sigma_m] \kappa (\beta_1 - 1)\) and return this value if \(n == 1\).

They use the \(\lbrack [\delta_1, \sigma_1, \delta_2, \sigma_2, \ldots, \delta_m, \sigma_m] \kappa\) prefix (se or smallerEmbed) which is the next smallest \(\text{nextLeast}\) Embeddings. See Section 11.2.2

(se.el == bel) is smallerEmbed.embedLevel == Embeddings::baseEmbedLevel

(se.el == bel) iff se is the prefix for an AdmisLevOrdinal.

dl is deleteLeast. hc is howCreated. nse is nestedSmallerEmbed, a version of se in class NestedEmbeddings.

(nse.hc == dl) is true iff se was created by deleting the least significant \(\delta, \sigma\) pair.

<table>
<thead>
<tr>
<th>X</th>
<th>Conditions</th>
<th>LimitTypeInfo</th>
<th>(\alpha.e(n))</th>
</tr>
</thead>
<tbody>
<tr>
<td>PLEB</td>
<td>(se.el == bel)</td>
<td>leastIndexSuccParam</td>
<td>for (i=1; i&lt;n; i++) b = ([\delta_1']\omega_{\kappa, b}.lp1); Rtn b</td>
</tr>
<tr>
<td>PLEC</td>
<td>se.el (!=) bel &amp; (nse.hc == dl)</td>
<td>leastIndexSuccParam</td>
<td>for (i=1; i&lt;n; i++) b = ([\delta_1', \delta_2, \sigma_2', \ldots, \delta_{m-1}, \sigma_{m-1}] \omega_{b}); Rtn b</td>
</tr>
<tr>
<td>PLED</td>
<td>se.el (!=) bel &amp; nse.hc (!=) dl</td>
<td>leastIndexSuccParam</td>
<td>for (i=1; i&lt;n; i++) b = ([\delta_1', \delta_2, \sigma_2', \ldots, \delta_{m-1}, \sigma_{m-1}, b] \omega_{b}); Rtn b</td>
</tr>
<tr>
<td>PLEE</td>
<td>se.el (!=) bel</td>
<td>leastLevelSuccParam</td>
<td>for (i=1; i&lt;n; i++) b = ([\delta_1', \delta_2, \sigma_2', \ldots, \delta_{m-2}, \sigma_{m-2}, \delta_{m-1}, b.lp1]) \omega_{\kappa}; Rtn b</td>
</tr>
</tbody>
</table>

See Table 64 for examples.

Table 60: AdmisNormalElement::paramLimitElement cases
**X** is an exit code (see Table 57). **UpX** is a higher level exit code from a calling routine.

<table>
<thead>
<tr>
<th>X</th>
<th>UpX</th>
<th>Ordinal</th>
<th>limitElement</th>
</tr>
</thead>
<tbody>
<tr>
<td>FL</td>
<td>NLA</td>
<td>([3,4,5] \omega_6(\omega_3))</td>
<td>([3,4,5] \omega_6(\omega_3[1] + 1))</td>
</tr>
<tr>
<td>IK</td>
<td>NLA</td>
<td>([4,12] \omega_{12,\omega_5}(5))</td>
<td>([4,12] \omega_{12,\omega_5[1]+1}[4,12] \omega_{12,\omega_5}(4)+1)</td>
</tr>
<tr>
<td>LCEL</td>
<td>NLA</td>
<td>([3,12] \omega_{\omega_3}[11])</td>
<td>([3,12] \omega_{\omega_3}[1]+3,[3,12] \omega_{\omega_3}(10)+1)</td>
</tr>
<tr>
<td>DQB</td>
<td>NLB</td>
<td>([3,4,1] \omega_1[1])</td>
<td>([3,4] \omega_4[3])</td>
</tr>
<tr>
<td>DQE</td>
<td>NLB</td>
<td>([3,4,4,1] \omega_4[4])</td>
<td>([3,4,4,1] \omega_4[3])</td>
</tr>
<tr>
<td>DCBO</td>
<td>NLC</td>
<td>([1,1] \omega_2[1])</td>
<td>([1,1] \omega_1[1])</td>
</tr>
<tr>
<td>DCBO</td>
<td>NLC</td>
<td>([1,3] \omega_3[1])</td>
<td>([1,3] \omega_3[1])</td>
</tr>
<tr>
<td>DCBO</td>
<td>NLC</td>
<td>([3,4,6] \omega_7[1])</td>
<td>([3,4,6] \omega_7[1])</td>
</tr>
<tr>
<td>DCAL</td>
<td>NLC</td>
<td>([2,12] \omega_{\omega+1}[\omega])</td>
<td>([2,12] \omega_{\omega+1}[1])</td>
</tr>
<tr>
<td>DCAL</td>
<td>NLC</td>
<td>([3,12] \omega_{\omega+1}[\omega])</td>
<td>([3,12] \omega_{\omega+1}[\omega][1])</td>
</tr>
<tr>
<td>DCAL</td>
<td>NLC</td>
<td>([10,5] \omega_{12}{1,2,3,1,2,3} \omega_2[1])</td>
<td>([10,5] \omega_{12}{1,2,3,1,2,3} \omega_2[1])</td>
</tr>
<tr>
<td>DCCS</td>
<td>NLC</td>
<td>([3,5] \omega_3[7])</td>
<td>([3,5] \omega_3[7])</td>
</tr>
<tr>
<td>NED</td>
<td>NLE</td>
<td>([3,12] \omega_{\omega+3})</td>
<td>([3,12] \omega_{\omega+3}[1]+3)</td>
</tr>
<tr>
<td>NEF</td>
<td>NLE</td>
<td>([4,5,1] \omega_5[5])</td>
<td>([4,5,1] \omega_5[5])</td>
</tr>
<tr>
<td>PLEC</td>
<td>NLP</td>
<td>([3,2,3,3] \omega_5(8))</td>
<td>([3,2,3,3] \omega_5(7))</td>
</tr>
<tr>
<td>PLED</td>
<td>NLP</td>
<td>([3,2,5,3] \omega_5(8))</td>
<td>([3,2,5,3] \omega_5(7))</td>
</tr>
<tr>
<td>PLEE</td>
<td>NLP</td>
<td>([5,\omega_3+4] \omega_{\omega+3}[9])</td>
<td>([5,\omega_3+4] \omega_{\omega+3}[8])</td>
</tr>
</tbody>
</table>

Table 61: NestedEmbedNormalElement::limitElement examples
<table>
<thead>
<tr>
<th>X</th>
<th>Ordinal</th>
<th>1</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>DQA</td>
<td>[5, 5, 1]</td>
<td>[5, 5, 1]</td>
<td>[5, 5, 1]</td>
</tr>
<tr>
<td>DQA</td>
<td>[1, 1, 1]</td>
<td>[1, 1]</td>
<td>[1, 1]</td>
</tr>
<tr>
<td>DQA</td>
<td>[5, 5, 1]</td>
<td>[5]</td>
<td>[5]</td>
</tr>
<tr>
<td>DQA</td>
<td>[12, 1]</td>
<td>[12]</td>
<td>[12]</td>
</tr>
<tr>
<td>DQA</td>
<td>[12, 1]</td>
<td>[12, 1]</td>
<td>[12, 1]</td>
</tr>
<tr>
<td>DQB</td>
<td>[12, 15, 15, 12]</td>
<td>[12, 15, 15, 11]</td>
<td>[12, 15, 15, 11]</td>
</tr>
<tr>
<td>DQB</td>
<td>[12, 15, 15]</td>
<td>[12, 15, 15]</td>
<td>[12, 15, 15]</td>
</tr>
<tr>
<td>DQB</td>
<td>[12, 15, 15]</td>
<td>[12, 15, 15]</td>
<td>[12, 15, 15]</td>
</tr>
<tr>
<td>DQB</td>
<td>[12, 15, 15, 12]</td>
<td>[12, 15, 15, 11]</td>
<td>[12, 15, 15, 11]</td>
</tr>
<tr>
<td>DQB</td>
<td>[12, 15, 15, 12]</td>
<td>[12, 15, 15, 11]</td>
<td>[12, 15, 15, 11]</td>
</tr>
<tr>
<td>DQB</td>
<td>[12, 15, 15, 12]</td>
<td>[12, 15, 15, 11]</td>
<td>[12, 15, 15, 11]</td>
</tr>
<tr>
<td>DQB</td>
<td>[12, 15, 15, 12]</td>
<td>[12, 15, 15, 11]</td>
<td>[12, 15, 15, 11]</td>
</tr>
<tr>
<td>DQB</td>
<td>[12, 15, 15, 12]</td>
<td>[12, 15, 15, 11]</td>
<td>[12, 15, 15, 11]</td>
</tr>
<tr>
<td>DQB</td>
<td>[12, 15, 15, 12]</td>
<td>[12, 15, 15, 11]</td>
<td>[12, 15, 15, 11]</td>
</tr>
<tr>
<td>DQB</td>
<td>[12, 15, 15, 12]</td>
<td>[12, 15, 15, 11]</td>
<td>[12, 15, 15, 11]</td>
</tr>
<tr>
<td>DQB</td>
<td>[12, 15, 15, 12]</td>
<td>[12, 15, 15, 11]</td>
<td>[12, 15, 15, 11]</td>
</tr>
<tr>
<td>DQB</td>
<td>[12, 15, 15, 12]</td>
<td>[12, 15, 15, 11]</td>
<td>[12, 15, 15, 11]</td>
</tr>
<tr>
<td>DQB</td>
<td>[12, 15, 15, 12]</td>
<td>[12, 15, 15, 11]</td>
<td>[12, 15, 15, 11]</td>
</tr>
<tr>
<td>DQB</td>
<td>[12, 15, 15, 12]</td>
<td>[12, 15, 15, 11]</td>
<td>[12, 15, 15, 11]</td>
</tr>
<tr>
<td>DQB</td>
<td>[12, 15, 15, 12]</td>
<td>[12, 15, 15, 11]</td>
<td>[12, 15, 15, 11]</td>
</tr>
<tr>
<td>DQB</td>
<td>[12, 15, 15, 12]</td>
<td>[12, 15, 15, 11]</td>
<td>[12, 15, 15, 11]</td>
</tr>
<tr>
<td>DQB</td>
<td>[12, 15, 15, 12]</td>
<td>[12, 15, 15, 11]</td>
<td>[12, 15, 15, 11]</td>
</tr>
<tr>
<td>DQB</td>
<td>[12, 15, 15, 12]</td>
<td>[12, 15, 15, 11]</td>
<td>[12, 15, 15, 11]</td>
</tr>
<tr>
<td>DQB</td>
<td>[12, 15, 15, 12]</td>
<td>[12, 15, 15, 11]</td>
<td>[12, 15, 15, 11]</td>
</tr>
<tr>
<td>DQB</td>
<td>[12, 15, 15, 12]</td>
<td>[12, 15, 15, 11]</td>
<td>[12, 15, 15, 11]</td>
</tr>
<tr>
<td>DQB</td>
<td>[12, 15, 15, 12]</td>
<td>[12, 15, 15, 11]</td>
<td>[12, 15, 15, 11]</td>
</tr>
<tr>
<td>DQB</td>
<td>[12, 15, 15, 12]</td>
<td>[12, 15, 15, 11]</td>
<td>[12, 15, 15, 11]</td>
</tr>
<tr>
<td>DQB</td>
<td>[12, 15, 15, 12]</td>
<td>[12, 15, 15, 11]</td>
<td>[12, 15, 15, 11]</td>
</tr>
<tr>
<td>DQB</td>
<td>[12, 15, 15, 12]</td>
<td>[12, 15, 15, 11]</td>
<td>[12, 15, 15, 11]</td>
</tr>
<tr>
<td>DQB</td>
<td>[12, 15, 15, 12]</td>
<td>[12, 15, 15, 11]</td>
<td>[12, 15, 15, 11]</td>
</tr>
<tr>
<td>DQB</td>
<td>[12, 15, 15, 12]</td>
<td>[12, 15, 15, 11]</td>
<td>[12, 15, 15, 11]</td>
</tr>
</tbody>
</table>

Table 62: NestedEmbedNormalElement::drillDownLimitElementEq examples
Table 63: NestedEmbedNormalElement::embedLimitElement examples

<table>
<thead>
<tr>
<th>X</th>
<th>Ordinal</th>
<th>1</th>
<th>limitElement</th>
</tr>
</thead>
<tbody>
<tr>
<td>NEA</td>
<td>[5, 12, 44, 19 \omega]_{25}</td>
<td>[5, 12, 44, 19 \omega]_{25}[1]</td>
<td>[5, 12, 44, 19 \omega]_{25}[2]</td>
</tr>
<tr>
<td>NEA</td>
<td>[12, 15]_{15}</td>
<td>[12, 15]_{15}[1]</td>
<td>[12, 15]_{15}[2]</td>
</tr>
<tr>
<td>NEA</td>
<td>[10, 15]_{20}</td>
<td>[10, 15]_{20}[1]</td>
<td>[10, 15]_{20}[2]</td>
</tr>
<tr>
<td>NEA</td>
<td>[12, 15]_{5}</td>
<td>[12, 15]_{5}[1]</td>
<td>[12, 15]_{5}[2]</td>
</tr>
<tr>
<td>NEB</td>
<td>[12, 4, 15, 11 \omega]_{15}^{(12)}</td>
<td>[12, 4, 15, 11 \omega]_{15}^{(11)+1}</td>
<td>[12, 4, 15, 11 \omega]_{15}^{(11)}</td>
</tr>
<tr>
<td>NEB</td>
<td>[12, 4, 15, 12 \omega]_{15}^{(12)+1}</td>
<td>[12, 4, 15, 12 \omega]_{15}^{(11)+1}</td>
<td>[12, 4, 15, 12 \omega]_{15}^{(11)+1}</td>
</tr>
<tr>
<td>NEC</td>
<td>[12, 4, 15, 13 \omega]_{15}^{(12)+1}</td>
<td>[12, 4, 15, 13 \omega]_{15}^{(11)+1}</td>
<td>[12, 4, 15, 13 \omega]_{15}^{(11)+1}</td>
</tr>
<tr>
<td>NED</td>
<td>[1, 2]<em>{12} \omega</em>{12}</td>
<td>[1, 2]<em>{12} \omega</em>{12}[1] + 1</td>
<td>[1, 2]<em>{12} \omega</em>{12}[2] + 1</td>
</tr>
<tr>
<td>NEF</td>
<td>[1, 1000, 1]<em>{3} \omega</em>{3}</td>
<td>[1, 1000, 1]<em>{3} \omega</em>{3}[1] + 1</td>
<td>[1, 1000, 1]<em>{3} \omega</em>{3}[2] + 1</td>
</tr>
<tr>
<td>NEF</td>
<td>[1, 1000, 1]<em>{3} \omega</em>{3}</td>
<td>[1, 1000, 1]<em>{3} \omega</em>{3}[1] + 1</td>
<td>[1, 1000, 1]<em>{3} \omega</em>{3}[2] + 1</td>
</tr>
<tr>
<td>NEF</td>
<td>[5, 12, 44, 19 \omega]_{19}^{(12)+1}</td>
<td>[5, 12, 44, 19 \omega]_{19}^{(11)+1}</td>
<td>[5, 12, 44, 19 \omega]_{19}^{(11)+1}</td>
</tr>
<tr>
<td>NEF</td>
<td>[5, 12, 44, 19 \omega]_{19}^{(12)+1}</td>
<td>[5, 12, 44, 19 \omega]_{19}^{(11)+1}</td>
<td>[5, 12, 44, 19 \omega]_{19}^{(11)+1}</td>
</tr>
<tr>
<td>NEF</td>
<td>[5, 12, 44, 19 \omega]_{19}^{(12)+1}</td>
<td>[5, 12, 44, 19 \omega]_{19}^{(11)+1}</td>
<td>[5, 12, 44, 19 \omega]_{19}^{(11)+1}</td>
</tr>
<tr>
<td>NEG</td>
<td>[2, 3]<em>{12} \omega</em>{12}</td>
<td>[2, 3]<em>{12} \omega</em>{12}[1] + 1</td>
<td>[2, 3]<em>{12} \omega</em>{12}[2] + 1</td>
</tr>
<tr>
<td>NEH</td>
<td>[12, 4, 15, 13 \omega]_{13}^{(12)+1}</td>
<td>[12, 4, 15, 13 \omega]_{13}^{(11)+1}</td>
<td>[12, 4, 15, 13 \omega]_{13}^{(11)+1}</td>
</tr>
</tbody>
</table>

Table 64: NestedEmbedNormalElement::paramLimitElement examples

<table>
<thead>
<tr>
<th>X</th>
<th>UpX</th>
<th>Ordinal</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>PLEB</td>
<td>NLP</td>
<td>[12, 1]<em>{12} \omega</em>{12}(3)</td>
<td>[12, 1]<em>{12} \omega</em>{12}(2)</td>
</tr>
<tr>
<td>PLEC</td>
<td>NLP</td>
<td>[5, 12, 44, 19 \omega]_{20}^{(12)+1}</td>
<td>[5, 12, 44, 19 \omega]_{20}^{(11)+1}</td>
</tr>
<tr>
<td>PLED</td>
<td>NLP</td>
<td>[12, 15, 1]_{15}^{(12)+1}</td>
<td>[12, 15, 1]_{15}^{(11)+1}</td>
</tr>
<tr>
<td>PLED</td>
<td>NLP</td>
<td>[12, 15, 1]_{15}^{(12)+1}</td>
<td>[12, 15, 1]_{15}^{(11)+1}</td>
</tr>
<tr>
<td>PLEE</td>
<td>NLP</td>
<td>[5, 12, 44, 19 \omega]_{19}^{(12)+1}</td>
<td>[5, 12, 44, 19 \omega]_{19}^{(11)+1}</td>
</tr>
<tr>
<td>LCDP</td>
<td>NLA</td>
<td>[12, 15]_{20}^{(12)+1}</td>
<td>[12, 15]_{20}^{(11)+1}</td>
</tr>
<tr>
<td>LCPD</td>
<td>NLA</td>
<td>[12, 15]_{20}^{(12)+1}</td>
<td>[12, 15]_{20}^{(11)+1}</td>
</tr>
</tbody>
</table>
X is an exit code (see Table 57). \textbf{UpX} is a higher level exit code from a calling routine.

<table>
<thead>
<tr>
<th>X</th>
<th>UpX</th>
<th>Ordinal</th>
<th>1</th>
<th>limitElement</th>
</tr>
</thead>
<tbody>
<tr>
<td>DDBO</td>
<td>LEAD</td>
<td>[1] (\omega_1)</td>
<td>(\omega_1)</td>
<td>(\omega_1) +1</td>
</tr>
<tr>
<td>DDCO</td>
<td>LEAD</td>
<td>[1] (\omega_1)</td>
<td>(\omega_1)</td>
<td>(\omega_1) +1</td>
</tr>
<tr>
<td>DCAL</td>
<td>LEAD</td>
<td>[4] (\omega_1)</td>
<td>(\omega_1)</td>
<td>(\omega_1) +1</td>
</tr>
<tr>
<td>DBO</td>
<td>LEAD</td>
<td>[1] (\omega_1)</td>
<td>(\omega_1)</td>
<td>(\omega_1) +1</td>
</tr>
<tr>
<td>DCCS</td>
<td>LEAD</td>
<td>[1] (\omega_1)</td>
<td>(\omega_1)</td>
<td>(\omega_1) +1</td>
</tr>
<tr>
<td>DBO</td>
<td>LEAD</td>
<td>[1] (\omega_1)</td>
<td>(\omega_1)</td>
<td>(\omega_1) +1</td>
</tr>
<tr>
<td>DQA</td>
<td>NLB</td>
<td>[1] (\omega_1)</td>
<td>(\omega_1)</td>
<td>(\omega_1) +1</td>
</tr>
<tr>
<td>DQA</td>
<td>NLB</td>
<td>[1] (\omega_1)</td>
<td>(\omega_1)</td>
<td>(\omega_1) +1</td>
</tr>
<tr>
<td>DQA</td>
<td>NLB</td>
<td>[1] (\omega_1)</td>
<td>(\omega_1)</td>
<td>(\omega_1) +1</td>
</tr>
<tr>
<td>DQA</td>
<td>NLB</td>
<td>[1] (\omega_1)</td>
<td>(\omega_1)</td>
<td>(\omega_1) +1</td>
</tr>
<tr>
<td>DQA</td>
<td>NLB</td>
<td>[1] (\omega_1)</td>
<td>(\omega_1)</td>
<td>(\omega_1) +1</td>
</tr>
<tr>
<td>DQA</td>
<td>NLB</td>
<td>[1] (\omega_1)</td>
<td>(\omega_1)</td>
<td>(\omega_1) +1</td>
</tr>
<tr>
<td>DQA</td>
<td>NLB</td>
<td>[1] (\omega_1)</td>
<td>(\omega_1)</td>
<td>(\omega_1) +1</td>
</tr>
<tr>
<td>DQA</td>
<td>NLB</td>
<td>[1] (\omega_1)</td>
<td>(\omega_1)</td>
<td>(\omega_1) +1</td>
</tr>
<tr>
<td>DQA</td>
<td>NLB</td>
<td>[1] (\omega_1)</td>
<td>(\omega_1)</td>
<td>(\omega_1) +1</td>
</tr>
<tr>
<td>DQA</td>
<td>NLB</td>
<td>[1] (\omega_1)</td>
<td>(\omega_1)</td>
<td>(\omega_1) +1</td>
</tr>
<tr>
<td>DQA</td>
<td>NLB</td>
<td>[1] (\omega_1)</td>
<td>(\omega_1)</td>
<td>(\omega_1) +1</td>
</tr>
<tr>
<td>DQA</td>
<td>NLB</td>
<td>[1] (\omega_1)</td>
<td>(\omega_1)</td>
<td>(\omega_1) +1</td>
</tr>
<tr>
<td>DQA</td>
<td>NLB</td>
<td>[1] (\omega_1)</td>
<td>(\omega_1)</td>
<td>(\omega_1) +1</td>
</tr>
</tbody>
</table>

Table 65: NestedEmbedNormalElement transitions
11.4 isValidLimitOrdParam and maxLimitType member functions

Neither of these routines are overridden at this code level. See Section 9.3 for a description of isValidLimitOrdParam and Section 9.5 for a description of maxLimitType.

11.5 limitInfo, limitType and embedType member functions

Only limitInfo overrides base class instances of these routines. NestedEmbedNormalElement::limitInfo computes the new types of limits and does most of the computation of computing limitType. See Table 52 for a description of this and sections 9.3 and 9.4 for more about all these routines.

11.6 limitOrd member function

As described in Section 9.6, NestedEmbedNormalElement::limitOrd extends the idea of limitElement as indirectly enumerating all smaller ordinals. It does this in a limited way for ordinals that are not recursive by using ordinal notations (including those yet to be defined) as arguments in place of the integer arguments of limitElement. By defining recursive operations on an incomplete domain we can retain something of the flavor of limitElement since: \( \alpha = \bigcup_{\beta} : \alpha.\text{isValidLimitOrdParam}(\beta) \alpha.\text{limitOrd}(\beta) \) and \( \alpha.\text{isValidLimitOrdParam}(\beta) \rightarrow \beta < \alpha \).

Table 66 gives the logic of NestedEmbedNormalElement::limitOrd along with the type of limit designated by LimitInfoType and the exit code used in debugging. Table 49 gives examples for each exit code. Table 68 gives values of enum LimitInfoType used by limitOrd in a case statement along with examples. Usually limitOrd is simpler than limitElement because it adds its argument as a \([\eta]\) or \([[[\eta]]]\) parameter to an existing parameter. Sometimes it must also decrement a successor ordinal and incorporate this in the result. The selection of which parameter(s) to modify is largely determined by a case statement on LimitInfoType. There are some additional examples in Table 51.

11.7 fixedPoint member function

The algorithm is similar to that in AdmisNormalElement::fixedPoint in Section 9.7. The two differences are that this static function accepts parameter a NestedEmbeddings instead of Embeddings and it has a final reference to a pointer to an ordinal that is used to return the value of the fixed point if one is found. i.e. if true is returned.

12 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. It replaces the imponderability 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...
\( \alpha \) is a notation for expression 13 with \( \eta = 0 \):

\[
\left[ [\delta_1, \sigma_1, \delta_2, \sigma_2, \ldots, \delta_m, \sigma_m] \omega_k \right] \text{ or } \left[ [\Delta_m] \omega_k \right]
\]

See tables 59 and 35 for symbol definitions.

<table>
<thead>
<tr>
<th>X</th>
<th>Info</th>
<th>( \alpha \text{limitOrd}(\zeta) )</th>
</tr>
</thead>
</table>
| NOA     | indexCKlimit                        | \[
[\delta_1, \sigma_1, \delta_2, \sigma_2, \ldots, \delta_m, \sigma_m] \omega_k, \text{loa}(\zeta)
\]                                |
| NOB     | leastIndexLimit                     | \[
[\delta_1, \sigma_1, \ldots, \delta_m-1, \sigma_m, \text{loa}(\zeta)] \omega_k
\]                                |
| NOC     | leastLevelLimit                     | \[
[\delta_1, \sigma_1, \ldots, \delta_{m-1}, \sigma_{m-1}, \delta_m, \text{loa}(\zeta)] \omega_{\delta_m, \text{loa}(\zeta)}
\]                                |
| NOD     | leastLevelLimitParam                | \[
\begin{align*}
\text{b} &= [\delta_1, \sigma_1, \delta_2, \sigma_2, \ldots, \delta_m, \sigma_m] \omega_k, (\beta_1 - 1); \text{ Rtn} \\
[\delta_1, \sigma_1, \ldots, \delta_{m-1}, \sigma_{m-1}, \delta_m, \text{loa}(\zeta)] \omega_{\delta_m, \text{loa}(\zeta), \text{b}, \text{lp1}}
\end{align*}
\]                                |
| NOE     | indexCKlimitParamEmbed              | \[
\begin{align*}
\text{b} &= [\delta_1, \sigma_1, \delta_2, \sigma_2, \ldots, \delta_m, \sigma_m] \omega_k, (\beta_1 - 1); \text{ Rtn} \\
[\delta_1, \sigma_1, \delta_2, \sigma_2, \ldots, \delta_m, \sigma_m] \omega_{\delta_m, \text{loa}(\zeta), b, \text{lp1}}
\end{align*}
\]                                |
| NOF     | leastIndexLimitParam                | \[
\begin{align*}
\text{b} &= [\delta_1, \sigma_1, \delta_2, \sigma_2, \ldots, \delta_m, \sigma_m] \omega_k, (\beta_1 - 1); \text{ Rtn} \\
[\delta_1, \sigma_1, \ldots, \delta_{m-1}, \sigma_{m-1}, \delta_m, \sigma_m, \text{loa}(\zeta)] \omega_{\delta_m, \text{b}, \text{lp1}}
\end{align*}
\]                                |
| NOG     | paramLimit                          | AdmisNormalElement::limitElementCom(n) |
|         | paramNxtLimit                       | See Table 48                           |
|         | functionLimit                       |                                       |
|         | functionNxtLimit                    |                                       |
|         | drillDownLimit                     |                                       |
|         | indexCKsuccEmbed                   |                                       |
|         | indexCKlimitEmbed                  |                                       |

See Table 67 for examples.

Table 66: NestedEmbedNormalElement::limitOrd cases
X is an exit code (see Table 66). **upX** is a higher level exit code from a calling routine.

<table>
<thead>
<tr>
<th>X</th>
<th>UpX</th>
<th>α</th>
<th>β</th>
<th>α.limitOrd(β)</th>
</tr>
</thead>
<tbody>
<tr>
<td>LOD</td>
<td>NOG</td>
<td>[2, 5, ω]ω5</td>
<td>ω</td>
<td>[2, 5, ω]ω5[ω][ω] + ω4</td>
</tr>
<tr>
<td>LOD</td>
<td>NOG</td>
<td>[ω12 + 1, ω15]ω1isiω12+1</td>
<td>ω1 + ω</td>
<td>[ω12 + 1, ω15]ω1isiω12+1 + ω15</td>
</tr>
<tr>
<td>LOD</td>
<td>NOG</td>
<td>[ω12 + 1, ω15]ω1isiω12+1</td>
<td>ω1 + ω12</td>
<td>[ω12 + 1, ω15]ω1isiω12+1ω12 + ω15</td>
</tr>
<tr>
<td>LOD</td>
<td>NOG</td>
<td>[2, 5, ω]ω5</td>
<td>ω</td>
<td>[2, 5, ω]ω5[ω][ω] + ω4</td>
</tr>
<tr>
<td>NOB</td>
<td>[3, 4, ω]ω5</td>
<td>ω</td>
<td></td>
<td>[3, 4, ω]ω5[ω][ω] + ω4</td>
</tr>
<tr>
<td>NOB</td>
<td>[3, 56, 5, 7, 8]ω8</td>
<td>ω1</td>
<td></td>
<td>[3, 56, 5, 7, 8]ω8[ω][ω] + 1</td>
</tr>
<tr>
<td>NOB</td>
<td>[3, 56, 5, 7, 8]ω8</td>
<td>ω1 + ω</td>
<td></td>
<td>[3, 56, 5, 7, 8]ω8[ω][ω] + ω</td>
</tr>
<tr>
<td>NOB</td>
<td>[3, 56, 5, 7, 8]ω8</td>
<td>ω1 + ω12</td>
<td></td>
<td>[3, 56, 5, 7, 8]ω8[ω][ω] + ω12</td>
</tr>
<tr>
<td>NOB</td>
<td>[4, 4, ω]ω12</td>
<td>ω</td>
<td></td>
<td>[4, 4, ω12]ω12[ω][ω] + ω4</td>
</tr>
<tr>
<td>NOB</td>
<td>[12, 5, 12]ω12</td>
<td>ω1</td>
<td></td>
<td>[12, 5, 12]ω12[ω][ω] + ω</td>
</tr>
<tr>
<td>NOB</td>
<td>[12, 5, 12]ω12</td>
<td>ω1 + ω12</td>
<td></td>
<td>[12, 5, 12]ω12[ω][ω] + ω12</td>
</tr>
<tr>
<td>LOF</td>
<td>NOG</td>
<td>[3, 1, 5]ω5</td>
<td>ω1</td>
<td>[3, 1, 5]ω5[ω][ω] + 1</td>
</tr>
<tr>
<td>LOF</td>
<td>NOG</td>
<td>[3, 1, 5]ω5</td>
<td>ω1 + ω</td>
<td>[3, 1, 5]ω5[ω][ω] + 1</td>
</tr>
<tr>
<td>LOF</td>
<td>NOG</td>
<td>[3, 1, 5]ω5</td>
<td>ω1 + ω12</td>
<td>[3, 1, 5]ω5[ω][ω] + 1</td>
</tr>
<tr>
<td>LOF</td>
<td>LOB</td>
<td>[3, 2]ω2</td>
<td>ω1</td>
<td>[3, 2ω2[ω][ω] + ω]ω2[ω][ω] + 3</td>
</tr>
<tr>
<td>LOF</td>
<td>LOB</td>
<td>[3, 2]ω2</td>
<td>ω1 + ω</td>
<td>[3, 2ω2[ω][ω] + ω]ω2[ω][ω] + 3</td>
</tr>
<tr>
<td>LOF</td>
<td>LOB</td>
<td>[3, 2]ω2</td>
<td>ω1 + ω12</td>
<td>[3, 2ω2[ω][ω] + ω]ω2[ω][ω] + 3</td>
</tr>
<tr>
<td>OFB</td>
<td>NOG</td>
<td>[4, 1, 5]ω5(ω4)</td>
<td>ω1</td>
<td>[4, 1, 5]ω5(ω4[ω][ω][ω][ω][ω][ω][ω][ω]] + 1</td>
</tr>
<tr>
<td>OFB</td>
<td>NOG</td>
<td>[4, 1, 5]ω5(ω4)</td>
<td>ω1 + ω</td>
<td>[4, 1, 5]ω5(ω4[ω][ω][ω][ω][ω][ω][ω][ω]][ω][ω][ω][ω][ω][ω][ω][ω][ω][ω][ω][ω][ω][ω][ω][ω][ω][ω][ω][ω][ω][ω][ω][ω][ω][ω][ω][ω][ω][ω][ω][ω][ω][ω][ω][ω][ω][ω][ω][ω][ω][ω][ω][ω][ω][ω][ω][ω][ω][ω][ω][ω][ω][ω][ω][ω][ω][ω][ω][ω][ω][ω][ω][ω][ω][ω][ω][ω][ω][ω][ω][ω][ω][ω][ω][ω][ω][ω][ω][ω][ω][ω][ω][ω][ω][ω][ω][ω][ω][ω][ω][ω][ω][ω][ω][ω][ω][ω][ω][ω][ω][ω][ω][ω][ω][ω][ω][ω][ω][ω][ω][ω][ω][ω][ω][ω][ω][ω][ω][ω][ω][ω][ω][ω][ω][ω][ω][ω][ω][ω][ω][ω][ω][ω][ω][ω][ω][ω][ω][ω][ω][ω][ω][ω][ω][ω][ω][ω][ω][ω][ω][ω][ω][ω][ω][ω][ω][ω][ω][ω][ω][ω][ω][ω][ω][ω][ω][ω][ω][ω][ω][ω][ω][ω][ω][ω][ω][ω][ω][ω][ω][ω][ω][ω][ω][ω][ω][ω][ω][ω][ω][ω][ω][ω][ω][ω][ω][ω][ω][ω][ω][ω][ω][ω][ω][ω][ω][ω][ω][ω][ω][ω][ω][ω][ω] + 1</td>
</tr>
<tr>
<td>OFB</td>
<td>NOG</td>
<td>[4, 1, 5]ω5(ω4)</td>
<td>ω1 + ω</td>
<td>[4, 1, 5]ω5(ω4[ω][ω][ω][ω][ω][ω][ω][ω][ω][ω][ω][ω][ω][ω][ω][ω][ω][ω][ω][ω][ω][ω][ω][ω][ω][ω][ω][ω][ω][ω][ω][ω][ω][ω][ω][ω][ω][ω][ω][ω][ω][ω][ω][ω][ω][ω][ω][ω][ω][ω][ω][ω][ω][ω][ω][ω][ω][ω][ω][ω][ω][ω][ω][ω][ω][ω][ω][ω][ω][ω][ω][ω][ω][ω][ω][ω][ω][ω][ω][ω][ω][ω][ω][ω][ω][ω][ω][ω][ω][ω][ω][ω][ω][ω][ω][ω][ω][ω][ω][ω][ω][ω][ω][ω][ω][ω][ω][ω][ω][ω][ω][ω][ω][ω][ω][ω][ω][ω][ω][ω][ω][ω][ω][ω][ω][ω][ω][ω][ω][ω][ω][ω][ω][ω][ω][ω][ω][ω][ω][ω][ω][ω][ω][ω][ω][ω][ω][ω][ω][ω][ω][ω][ω][ω][ω][ω][ω][ω][ω][ω][ω][ω][ω][ω][ω][ω][ω][ω][ω][ω][ω][ω][ω][ω][ω][ω][ω][ω][ω][ω][ω][ω][ω][ω][ω][ω][ω][ω][ω][ω][ω][ω][ω][ω][ω][ω][ω][ω][ω][ω][ω][ω][ω][ω][ω][ω][ω][ω][ω][ω][ω][ω][ω][ω][ω][ω][ω][ω][ω][ω][ω][ω][ω][ω][ω][ω][ω][ω][ω][ω][ω][ω][ω] + 1</td>
</tr>
</tbody>
</table>

Table 67: NestedEmbedNormalElement::limitOrd examples
be done, from outside the system, by a Cantor diagonalization of the reals definable within the system.

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

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

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

Since all ordinals beyond the integers are infinite they do not correspond to anything in the physical world. Our idea of all integers comes from the idea that we can define what an integer is. The property of being an integer leads to the idea that there is a set of all objects satisfying the property. An alternative way to think of the integers is computationally. We can write a computer program that can in theory output every integer. Of course real programs do not run forever, error free, but that does not mean that such potentially infinite operations as a computer running forever lack physical significance. Our universe appears to be extraordinarily large, but finite. However, it might be potentially infinite. Cosmology is of necessity a speculative science. Thus the idea of a potentially infinite set of all integers, in contrast to that of completed infinite totalities, might have objective meaning in physical reality. Most of standard mathematics has an interpretation in an always finite but potentially infinite universe, but some questions, such as the continuum hypothesis do not. This meshes with increasing skepticism about whether the continuum hypothesis and other similar foundations questions are objectively true or false as Solomon Feferman and others have suggested[8].

Appendix A is a first attempt at translating these ideas into a restricted formalization of ZF. This appendix provides additional disussion of the philosophy underlying this approach.

Subsets of the integers are the G"odel numbers of TMs that satisfy a logically determined property.
A  Formalizing objective mathematics

A.1  Introduction

Objective mathematics attempts to distinguish between statements that are objectively true or false and those that are only true, false or undecidable relative to a particular formal system. This distinction is based on the assumption of an always finite but perhaps potentially infinite universe\(^{52}\). This is a return to the earlier conception of mathematical infinity as a potential that can never be realized. This is not to ignore the importance and value of the algebra of the infinite that has grown out of Cantor’s approach to mathematical infinity. It does suggest a reinterpretation of those results in terms of the countable models implied by the Löwenheim-Skolem theorem. It also suggests approaches for expanding the foundations of mathematics that include using the computer as a fundamental research tool. These approaches may be more successful at gaining wide spread acceptance than large cardinal axioms which are far removed from anything physically realizable.

A.1.1  The mathematics of recursive processes

A core idea is that only the mathematics of finite structures and properties of recursive processes is objective. This does not include uncountable sets, but it does include much of mathematics including some statements that require quantification over the reals\(^4\). For example, the question of whether a recursive process defines a notation for a recursive ordinal requires quantification over the reals to state but is objective.

Loosely speaking objective properties of recursive processes are those logically determined by a recursively enumerable sequence of events. This cannot be precisely formulated, but one can precisely state which set definitions in a formal system meet this criteria (see Section A.7).

The idea of objective mathematics is closely connected to generalized recursion theory. The latter starts with recursive relations and expands these with quantifiers over the integers and reals. As long as the relations between the quantified variable are recursive, the events that logically determine the result are recursively enumerable.

A.1.2  The uncountable

It is with the uncountable that contemporary set theory becomes incompatible with infinity as a potential that can never be realized. Proving something is true for all entities that meet some property does not require that a collection of all objects that satisfy that property exists. Real numbers exist as potentially infinite sequences that are either recursively enumerable or defined by a non computable, but still logically determined, mathematical expression. The idea of the collection of all reals is closely connected with Cantor’s proof that the reals are not countable. For that proof to work reals must exist as completed infinite decimal expansions or some logically equivalent structure. This requires infinity as an actuality and not just a potential.

\(^{52}\)A potentially infinite universe is one containing a finite amount of information at any point in time but with unbounded growth over time in its information content.
This appendix is a first attempt to formally define which statements in Zermelo Frankel set theory (ZF)[6] are objective. The goal is not to offer a weaker alternative. ZF has an objective interpretation in which all objective questions it decides are correctly decided. The purpose is to offer a new interpretation of the theory that seems more consistent with physical reality as we know it. This interpretation is relevant to extending mathematics. Objective questions have a truth value independent of any formal system. If they are undecidable in existing axiom systems, one might search for new axioms to decide them. In contrast there is no basis on which relative questions, like the continuum hypothesis, can be objectively decided.

A.1.3 Expanding the foundations of mathematics

Defining objective mathematics may help to shift the focus for expanding mathematics away from large cardinal axioms. Perhaps in part because they are not objective, it has not been possible to reach consensus about using these to expand mathematics. An objective alternative is to expand the hierarchy of recursive and countable ordinals by using computers to deal with the combinatorial explosion that results[5].

Throughout the history of mathematics, the nature of the infinite has been a point of contention. There have been other attempts to make related distinctions. Most notable is intuitionism stated by Brouwer[1]. These approaches can involve (and intuitionism does involve) weaker formal systems that allow fewer questions to be decidable and with more difficult proofs. Mathematicians consider Brouwer’s approach interesting and even important but few want to be constrained by its limitations. A long term aim of the approach of this appendix is to define a formal system that is widely accepted and is stronger than ZF in deciding objective mathematics.

A.2 Background

The most widely used formalization of mathematics, Zermelo Frankel set theory plus the axiom of choice (ZFC)[6], gives the same existential status to every object from the empty set to large cardinals. Finite objects and structures can exist physically. As far as we know this is not true of any infinite objects. Our universe could be potentially infinite but it does not seem to harbor actual infinities.

This disconnect between physical reality and mathematics has long been a point of contention. One major reason it has not been resolved is the power of the existing mathematical framework to solve problems that are relevant to a finite but potentially infinite universe and that cannot be solved by weaker systems. A field of mathematics has been created, called reverse mathematics, to determine the weakest formal system that can solve specific problems. There are problems in objective mathematics that have been shown to be solvable only by using large cardinal axioms that extend ZF. This however has not resulted in widespread acceptance of such axioms. For one thing there are weaker axioms (in terms of definability) that could solve these problems. There do not exist formal systems limited to objective mathematics that include such axioms[53] in part because of the combinatorial complexity.

[53]Axioms that assert the existence of large recursive ordinals can provide objective extensions to objective formalizations of mathematics. Large cardinal axioms imply the existence of large recursive ordinals that
they require. Large cardinal axioms are a simpler and more elegant way to accomplish the same result, but one can prove that alternatives exist. Mathematics can be expanded at many levels in the ordinal hierarchy. Determining the minimal ordinal that decides some question is different from determining the minimum formal system that does so.

### A.2.1 The ordinal hierarchy

Ordinal numbers generalize induction on the integers. As such they form the backbone of mathematics. Every integer is an ordinal. The set of all integers, $\omega$, is the smallest infinite ordinal. There are three types of ordinals: 0 (or the empty set), successors and limits. $\omega$ is the first limit ordinal. The successor of an ordinal is defined to be the union of the ordinal with itself. Thus for any two ordinals $a$ and $b$ $a < b \equiv a \in b$. This is very convenient, but it masks the rich combinatorial structure required to define finite ordinal notations and the rules for manipulating them.

From an objective standpoint it is more useful to think of ordinals as properties of recursive processes. The recursive ordinals are those whose structure can be enumerated by a recursive process. For any recursive ordinal, $R$, one can define a unique sequence of finite symbols (a notation) to represent each ordinal $\leq R$. For these notations one can define a recursive process that evaluates the relative size of any two notations and a recursive process that enumerates the notations for all ordinals smaller than that represented by any notation.

Starting with the recursive ordinals there are many places where the hierarchy can be expanded. It appears that the higher up the ordinal hierarchy one works, the stronger the results that can be obtained for a given level of effort. However, I suspect, and history suggests, that the strongest results will ultimately be obtained by working out the details at the level of recursive and countable ordinals. These are the objective levels.

### A.3 The true power set

Going beyond the countable ordinals with the power set axiom moves beyond objective mathematics. No formal system can capture what mathematicians want to mean by the true power set of the integers or any other uncountable set. This follows from Cantor’s proof that the reals are not countable and the Löwenheim-Skolem theorem that established that every formal system that has a model must have a countable model. The collection of all the subsets of the integers provably definable in ZF is countable. Of course it is not countable within ZF. The union of all sets provably definable by any large cardinal axiom defined now or that ever can be defined in any possible finite formal system is countable. One way some mathematicians claim to get around this is to say the true ZF includes an axiom for every true real number asserting its existence. This is a bit like the legislator who wanted to pass a law that $\pi$ is $3 \, \frac{1}{7}$. You can make the law but you cannot enforce it.

My objections to ZF are not to the use of large cardinal axioms, but to some of the philosophical positions associated with them and the practical implications of those positions.\textsuperscript{3} One way some mathematicians claim to get around this is to say the true ZF includes an axiom for every true real number asserting its existence. This is a bit like the legislator who wanted to pass a law that $\pi$ is $3 \, \frac{1}{7}$. You can make the law but you cannot enforce it.

My objections to ZF are not to the use of large cardinal axioms, but to some of the philosophical positions associated with them and the practical implications of those positions.\textsuperscript{3}
Instead of seeing formal systems for what they are, recursive processes for enumerating theorems, they are seen by some as transcending the finite limits of physical existence. In the Platonic philosophy of mathematics, the human mind transcends the limitations of physical existence with direct insight into the nature of the infinite. The infinite is not a potential that can never be realized. It is a Platonic objective reality that the human mind, when properly trained, can have direct insight into.

This raises the status of the human mind and, most importantly, forces non mental tools that mathematicians might use into a secondary role. This was demonstrated when a computer was used to solve the long standing four color problem because of the large number of special cases that had to be considered. Instead of seeing this as a mathematical triumph that pointed the way to leveraging computer technology to aid mathematics, there were attempts to delegitimize this approach because it went beyond what was practical to do by human mental capacity alone.

Computer technology can help to deal with the combinatorial explosion that occurs in directly developing axioms for large recursive ordinals\(^5\). Spelling out the structure of these ordinals is likely to provide critical insight that allows much larger expansion of the ordinal hierarchy than is possible with the unaided human mind even with large cardinal axioms. If computers come to play a central role in expanding the foundations of mathematics, it will significantly alter practice and training in some parts of mathematics.

### A.4 Mathematical Objects

In the philosophical framework of this appendix there are three types of mathematical ‘objects’:

1. finite sets,
2. properties of finite sets and
3. properties of properties.

Finite sets are abstract idealizations of what can exist physically.

Objective properties of finite sets, like being an even integer, are logically determined human creations. Whether a particular finite set has the property follows logically from the definition of the property. However, the property can involve questions that are logically determined but not determinable. One example is the set of Gödel numbers of Turing Machines (TM)\(^5\) that do not halt. What the TM does at each time step is logically determined and thus so is the question of whether it will halt, but, if it does not halt, there is no general way to determine this. The non-halting property is objectively determined but not in general determinable.

\(^5\)There is overwhelming evidence that one can create a Universal Turing Machine that can simulate every possible computer. Assuming this is true, one can assign a unique integer to every possible program for this universal computer. This is called a Gödel number because Gödel invented this idea in a different, but related, context a few years before the concept of a Universal Turing Machine was proposed.
A.4.1 Properties of properties

The property of being a subset of the integers has led to the idea that uncountable collections exist. No finite or countable formal system can capture what mathematicians want to mean by the set of all subsets of the integers. One can interpret this as the human mind transcending the finite or that mathematics is a human creation that can always be expanded. The inherent incompleteness of any sufficiently strong formal system similarly suggests that mathematics is creative.

A.4.2 Gödel and mathematical creativity

Gödel proved that any formalization of mathematics strong enough to embed the primitive recursive functions (or alternatively a Universal Turing Machine) must be either incomplete or inconsistent[7]. In particular such a system, will be able to model itself, and will not be able to decide if this model of itself is consistent unless it is inconsistent.

This is often seen as putting a limit on mathematical knowledge. It limits what we can be certain about in mathematics, but not what we can explore. A divergent creative process can, in theory, pursue every possible finite formalization of mathematics as long as it does not have to choose which approach or approaches are correct. Of course it can rule out those that are discovered to be inconsistent or to have other provable flaws. This may seem to be only of theoretical interest. However the mathematically capable human mind is the product of just such a divergent creative process known as biological evolution.

A.4.3 Cantor’s incompleteness proof

One can think of Cantor’s proof (when combined with the Löwenheim-Skolem theorem) as the first great incompleteness proof. He proved the properties defining real numbers can always be expanded. The standard claim is that Cantor proved that there are “more” reals than integers. That claim depends on each real existing as a completed infinite totality. From an objective standpoint, Cantor’s proof shows that any formal system that meets certain prerequisites can be expanded by diagonalizing the real numbers provably definable within the system. Of course this can only be done from outside the system.

Just as one can always define more real numbers one can always create more objective mathematics. One wants to include as objective all statements logically determined by a recursively enumerable sequence of events, but that can only be precisely defined relative to a particular formal system and will always be incomplete and expandable just as the reals are.

A.5 Axioms of ZFC

The formalization of objectivity starts with the axioms of Zermelo Frankel Set Theory plus the axiom of choice ZFC, the most widely used formalization of mathematics. This is not the ideal starting point for formalizing objective mathematics but it is the best approach to clarify where in the existing mathematical hierarchy objective mathematics ends. To that end a restricted version of these axioms will be used to define an objective formalization of mathematics.
The following axioms are adapted from *Set Theory and the Continuum Hypothesis*.[6]

A.5.1 Axiom of extensionality

Without the axiom that defines when two sets are identical (=) there would be little point in defining the integers or anything else. The axiom of extensionality says sets are uniquely defined by their members.

\[ \forall x \forall y \ (\forall z \ z \in x \equiv z \in y) \equiv (x = y) \]

This axiom[56] says a pair of sets \( x \) and \( y \) are equal if and only if they have exactly the same members.

A.5.2 Axiom of the empty set

The empty set must be defined before any other set can be defined.

The axiom of the empty set is as follows.

\[ \exists x \forall y \neg (y \in x) \]

This axiom[57] says there exists an object \( x \) that no other set belongs to. \( x \) contains nothing. The empty set is denoted by the symbol \( \emptyset \).

A.5.3 Axiom of unordered pairs

From any two sets \( x \) and \( y \) one can construct a set that contains both \( x \) and \( y \). The notation for that set is \( \{x, y\} \).

\[ \forall x \forall y \exists z \forall w \ w \in z \equiv (w = x \lor w = y) \]

This says for every pair of sets \( x \) and \( y \) there exists a set \( w \) that contains \( x \) and \( y \) and no other members. This is written as \( w = x \cup y \).

A.5.4 Axiom of union

A set is an arbitrary collection of objects. The axiom of union allows one to combine the objects in many different sets and make them members of a single new set. It says one can go down two levels taking not the members of a set, but the members of members of a set and combine them into a new set.

\[ \forall x \exists y \forall z \ z \in y \equiv (\exists t \ z \in t \land t \in x) \]

[55]The axioms use the existential quantifier (\( \exists \)) and the universal quantifier (\( \forall \)). \( \exists x g(x) \) means there exists some set \( x \) for which \( g(x) \) is true. Here \( g(x) \) is any expression that includes \( x \). \( \forall x g(x) \) means \( g(x) \) is true of every set \( x \).

[56]\( a \equiv b \) means \( a \) and \( b \) have the same truth value or are equivalent. They are either both true or both false. It is the same as \( (a \rightarrow b) \land (b \rightarrow a) \).

[57]The ‘\( \neg \)’ symbol says what follow is not true.
This says for every set $x$ there exists a set $y$ that is the union of all the members of $x$. Specifically, for every $z$ that belongs to the union set $y$ there must be some set $t$ such that $t$ belongs to $x$ and $z$ belongs to $t$.

A.5.5 Axiom of infinity

The integers are defined by an axiom that asserts the existence of a set $\omega$ that contains all the integers. $\omega$ is defined as the set containing 0 and having the property that if $n$ is in $\omega$ then $n + 1$ is in $\omega$. From any set $x$ one can construct a set containing $x$ by constructing the unordered pair of $x$ and $x$. This set is written as $\{x\}$.

$$\exists x \emptyset \in x \land [\forall y (y \in x) \rightarrow (y \cup \{y\} \in x)]$$

This says there exists a set $x$ that contains the empty set $\emptyset$ and for every set $y$ that belongs to $x$ the set $y + 1$ constructed as $y \cup \{y\}$ also belongs to $x$.

The axiom of infinity implies the principle of induction on the integers.

A.5.6 Axiom scheme of replacement

The axiom scheme for building up complex sets like the ordinals is called replacement. They are an easily generated recursively enumerable infinite sequence of axioms.

The axiom of replacement scheme describes how new sets can be defined from existing sets using any relationship $A_n(x, y)$ that defines $y$ as a function of $x$. A function maps any element in its range (any input value) to a unique result or output value.

$\exists! y \ g(y)$ means there exists one and only one set $y$ such that $g(y)$ is true. The axiom of replacement scheme is as follows.

$$B(u, v) \equiv [\forall y (y \in v \equiv \exists x [x \in u \land A_n(x, y)]]$$

$$[\forall x \exists! y A_n(x, y) \rightarrow \forall u \exists v (B(u, v))]$$

That first line defines $B(u, v)$ as equivalent to $y \in v$ if and only if there exists an $x \in u$ such that $A_n(x, y)$ is true. One can think of $A_n(x, y)$ as defining a function that may have multiple values for the same input. $B(u, v)$ says $v$ is the image of $u$ under this function.

This second line says if $A_n$ defines $y$ uniquely as a function of $x$ then the for all $u$ there exists $v$ such that $B(u, v)$ is true.

This axioms says that, if $A_n(x, y)$ defines $y$ uniquely as a function of $x$, then one can take any set $u$ and construct a new set $v$ by applying this function to every element of $u$ and taking the union of the resulting sets.

This axiom schema came about because previous attempts to formalize mathematics were too general and led to contradictions like the Barber Paradox\textsuperscript{58}. By restricting new sets to those obtained by applying well defined functions to the elements of existing sets it was felt that one could avoid such contradictions. Sets are explicitly built up from sets defined in safe axioms. Sets cannot be defined as the universe of all objects satisfying some relationship. One cannot construct the set of all sets which inevitably leads to a paradox.

\textsuperscript{58} The barber paradox concerns a barber who shaves everyone in the town except those who shave themselves. If the barber shaves himself then he must be among the exceptions and cannot shave himself. Such a barber cannot exist.
A.5.7 Power set axiom

The power set axiom says the set of all subsets of any set exists. This is not needed for finite sets, but it is essential to define the set of all subsets of the integers.

\[ \forall x \exists y \forall z [z \in y \equiv z \subseteq x] \]

This says for every set \( x \) there exists a set \( y \) that contains all the subsets of \( x \). \( z \) is a subset of \( x \) (\( z \subseteq x \)) if every element of \( z \) is an element of \( x \).

The axiom of the power set completes the axioms of ZF or Zermelo Frankel set theory. From the power set axiom one can conclude that the set of all subsets of the integers exists. From this set one can construct the real numbers.

This axioms is necessary for defining recursive ordinals which is part of objective mathematics. At the same time it allows for questions like the continuum hypothesis that are relative. Drawing the line between objective and relative properties is tricky.

A.5.8 Axiom of Choice

The Axiom of Choice is not part of ZF. It is however widely accepted and critical to some proofs. The combination of this axiom and the others in ZF is called ZFC.

The axiom states that for any collection of non empty sets \( C \) there exists a choice function \( f \) that can select an element from every member of \( C \). In other words for every \( e \in C \) \( f(e) \in e \).

\[ \forall C \exists f \forall e [(e \in C \land e \neq \emptyset) \rightarrow f(e) \in e] \]

A.6 The axioms of ZFC summary

1. Axiom of extensionality (See Section A.5.1).

\[ \forall x \forall y (\forall z \ z \in x \equiv z \in y) \equiv (x = y) \]

2. Axiom of the empty set (See Section A.5.2).

\[ \exists x \forall y \neg(y \in x) \]

3. Axiom of unordered pairs (See Section A.5.3).

\[ \forall x \forall y \exists z \forall w \ w \in z \equiv (w = x \lor w = y) \]

4. Axiom of union (See Section A.5.4).

\[ \forall x \exists y \forall z \ z \in y \equiv (\exists t \ z \in t \land t \in x) \]

5. Axiom of infinity (See Section A.5.5).

\[ \exists x \emptyset \in x \land [\forall y (y \in x) \rightarrow (y \cup \{y\} \in x)] \]
6. Axiom schema of replacement (See Section A.5.6).

\[ B(u, v) \equiv [\forall y (y \in v \equiv \exists x [x \in u \land A_n(x, y)])] \]

\[ [\forall x \exists ! y A_n(x, y)] \rightarrow \forall u \exists v(B(u, v)) \]

7. Axiom of the power set (See Section A.5.7).

\[ \forall x \exists y \forall z [z \in y \equiv z \subseteq x] \]

8. Axiom of choice (See Section A.5.8).

\[ \forall C \exists f \forall e [(e \in C \land e \neq \emptyset) \rightarrow f(e) \in e] \]

## A.7 The Objective Parts of ZF

Objectivity is a property of set definitions. Its domain is expressions within ZF (or any formalization of mathematics) that define new sets (or other mathematical objects). A set is said to be objective if it can be defined by an objective statement.

The axiom of the empty set and the axiom of infinity are objective. The axiom of unordered pairs and the axiom of union are objective when they define new sets using only objective sets. The power set axiom applied to an infinite set is not objective and it is unnecessary for finite sets.

A limited version of the axiom of replacement is objective. In this version the formulas that define functions (the \(A_n\) in this appendix) are limited to recursive relations on the bound variables and objective constants. Both universal and existential quantifiers are limited to ranging over the integers or subsets of the integers. Without the power set axiom, the subsets of the integers do not form a set. However the property of being a subset of the integers

\[ S(x) \equiv \forall y y \in x \rightarrow y \in \omega \]

can be used to restrict a bound variable.

Quantifying over subsets of the integers suggests searching through an uncountable number of sets. However, by only allowing a recursive relation between bound variables and objective constants, one can enumerate all the events that determine the outcome. A computer program that implements a recursive relationship on a finite number of subsets of the integers must do a finite number of finite tests so the result can be produced in a finite time. A nondeterministic computer program can enumerate all of these results. For example the formula

\[ \forall r S(r) \rightarrow \exists n (n \in \omega \rightarrow a(r, n)) \]

\[ 59 \]

In this context nondeterministic refers to a computer that simulates many other computer programs by emulating each of them and switching in time between them in such a way that every program is fully executed. The emulation of a program stops only if the emulated program halts. The programs being emulated must be finite or recursively enumerable. In this context nondeterministic does not mean unpredictable.
is determined by what a recursive process does for every finite initial segment of every subset of the integers. One might think of this approach as a few steps removed from constructivism. One does not need to produce a constructive proof that a set exists. One does need to prove that every event that determines the members of the set is constructible.

A.8 Formalization of the Objective Parts of ZF

Following are axioms that define the objective parts of ZF as outlined in the previous section. The purpose is not to offer a weaker alternative to ZF but to distinguish the objective and relative parts of that system.

A.8.1 Axioms unchanged from ZF

As long as the universe of all sets is restricted to objective sets the following axioms are unchanged from ZF.

1. Axiom of extensionality
   \[ \forall x \forall y \ (\forall z \ z \in x \equiv z \in y) \equiv (x = y) \]

2. Axiom of the empty set
   \[ \exists x \forall y \ \neg (y \in x) \]

3. Axiom of unordered pairs
   \[ \forall x \forall y \exists z \forall w \ w \in z \equiv (w = x \lor w = y) \]

4. Axiom of union
   \[ \forall x \exists y \forall z \ z \in y \equiv (\exists t \ z \in t \land t \in x) \]

5. Axiom of infinity
   \[ \exists x \emptyset \in x \land [\forall y (y \in x) \rightarrow (y \cup \{y\} \in x)] \]

A.8.2 Objective axiom of replacement

In the following \( A_n \) is any recursive relation in the language of ZF in which constants are objectively defined and quantifiers are restricted to range over the integers (\( \omega \)) or be restricted to subsets of the integers. Aside from these restrictions on \( A_n \), the objective active of replacement is the same as it is in ZF.

\[ B(u, v) \equiv [\forall y (y \in v \equiv \exists x [x \in u \land A_n(x, y)])] \]
\[ [\forall x \exists ! y A_n(x, y)] \rightarrow \forall u \exists v (B(u, v)) \]

\(^{60}\)Initial segments of subsets of the integers are ordered and thus defined by the size of the integers.
A.9 An Objective Interpretation of ZFC

I suspect it is consistent to assume the power set axiom in ZF, because all subsets of the integers (and larger cardinals) that are provably definable in ZF form a definite, albeit countable, collection. These are definite collections only relative to a specific formal system. Expand ZF with an axiom like “there exists an inaccessible cardinal” and these collections expand.

Uncountable sets in ZF suggest how the objective parts of ZF can be expanded. Create an explicitly countable definition of the countable ordinals defined by the ordinals that are uncountable within ZF. Expand ZF to ZF+ with axioms that assert the existence of these structures. This approach to expansion can be repeated with ZF+. The procedure can be iterated and it must have a fixed point that is unreachable with these iterations.

Ordinal collapsing functions do something like this. They use uncountable ordinals as notations for recursive ordinals to expand the recursive ordinals. Ordinal collapsing can also use countable ordinals larger than the recursive ordinals. This is possible at multiple places in the ordinal hierarchy. I suspect that uncountable ordinals provide a relatively weak way to expand the recursive and larger countable ordinal hierarchies. The countable ordinal hierarchy is a bit like the Mandelbrot set. The hierarchy definable in any particular formal system can be embedded within itself at many places.

The objective interpretation of ZFC see it as a recursive process for defining finite sets, properties of finite sets and properties of properties. These exist either as physical objects that embody the structure of finite sets or as expressions in a formal language that can be connected to finite objects and/or expressions that define properties. Names of all the objects that provably satisfy the definition of any set in ZF are recursively enumerable because all proofs in any formal system are. These names and their relationships form an interpretation of ZF.

A.10 A Creative Philosophy of Mathematics

Platonic philosophy visualizes an ideal realm of absolute truth and beauty of which the physical world is a dim reflection. This ideal reality is perfect, complete and thus static. In stark contrast, the universe we inhabit is spectacularly creative. An almost amorphous cosmic big bang has evolved into an immense universe of galaxies each of which is of a size and complexity that takes ones breath away. On at least one minuscule part of one of the these galaxies, reproducing molecules have evolved into the depth and richness of human conscious experience. There is no reason to think that we at a limit of this creative process. There may be no finite limit to the evolution of physical structure and the evolution of consciousness. This is what the history of the universe, this planet and the facts of mathematics suggest to me. We need a new philosophy of mathematics grounded in our scientific understanding and the creativity that mathematics itself suggests is central to both developing mathematics and the content created in doing do.

Mathematics is both objective and creative. If a TM runs forever, this is logically determined by its program. Yet it takes creativity to develop a mathematical system to prove this. Gödel proved that no formal system that is sufficiently strong can be complete, but there is nothing (except resources) to prevent an exploration over time of every possible formalization of mathematics. As mentioned earlier, it is just such a process that created
the mathematically capable human mind. The immense diversity of biological evolution was probably a necessary prerequisite for evolving that mind.

Our species has a capacity for mathematics as a genetic heritage. We will eventually exhaust what we can understand from exploiting that biological legacy through cultural evolution. This exhaustion will not occur as an event but a process that keeps making progress. However there must be a Gödel limit to the entire process even if it continues forever. Following a single path of mathematical development will lead to an infinite sequence of results all of which are encompassed in a single axiom that will never be explored. This axiom will only be explored if mathematics becomes sufficiently diverse. In the long run, the only way to avoid a Gödel limit to mathematical creativity is through ever expanding diversity.

There is a mathematics of creativity that can guide us in pursuing diversity. Loosely speaking the boundary between the mathematics of convergent processes and that of divergent creative processes is the Church-Kleene ordinal or the ordinal of the recursive ordinals. For every recursive ordinal \( r_0 \) there is a recursive ordinal \( r_1 (r_0 \leq r_1) \) such that there are halting problems decidable by \( r_1 \) and not by any smaller ordinal. In turn every halting problem is decidable by some recursive ordinal. The recursive ordinals can decide the objective mathematics of convergent or finite path processes. Larger countable ordinals define a mathematics of divergent processes, like biological evolution, that follow an ever expanding number of paths.\(^{61}\)

The structure of biological evolution can be connected to a divergent recursive process. To illustrate this consider a TM that has an indefinite sequence of outputs that are either terminal nodes or the Gödel numbers of other recursive processes. In the latter case the TM that corresponds to the output must have its program executed and its outputs similarly interpreted. A path is a sequence of integers that corresponds to the output index at each level in the simulation hierarchy. For example the initial path segment \((4,1,3)\) indexes a path that corresponds to the fourth output of the root TM \((r_4)\), the first output of \(r_4 (r_4,1)\) and the third output of \(r_{4,1} (r_{4,1,3})\). These paths have the structure of the tree of life that shows what species were descended from which other species.

Questions about divergent recursive processes can be of interest to inhabitants of an always finite but potentially infinite universe. For example one might want to know if a given species will evolve an infinite chain of descendant species. In a deterministic universe, this problem can be stated using divergent recursive processes to model species. We evolved through a divergent creative process that might or might not be recursive. Quantum mechanics implies that there are random perturbations, but that may not be the final word.

Even with random perturbations, questions about all the paths a divergent recursive process can follow, may be connected to biological and human creativity. Understanding these processes may become increasingly important in the next few decades as we learn to control and direct biological evolution. Today there is intense research on using genetic engineering to cure horrible diseases. In time these techniques will become safe, reliable and predictable. The range of applications will inevitably expand. At that point it will become

\[^{61}\text{In a finite universe there are no truly divergent processes. Biological evolution can be truly divergent only if our universe is potentially infinite and life on earth migrates to other planets, solar systems and eventually galaxies.}\]
extremely important to have as deep an understanding as possible of what we may be doing. To learn more about this see[3].

B  Command line interface

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

B.1  Introduction

The Ordinal Calculator is an interactive tool for understanding the hierarchies of recursive and countable ordinals[18, 14, 9]. It is also a research tool to help to expand these hierarchies. Its motivating goal is ultimately to expand the foundations of mathematics by using computer technology to manage the combinatorial explosion in complexity that comes with explicitly defining the recursive ordinals implicitly defined by the axioms of Zermelo-Frankel set theory[6, 3]. The underlying philosophy focuses on what formal systems tell us about physically realizable combinatorial processes[4].

The source code and documentation is licensed for use and distribution under the Gnu General Public License, Version 2, June 1991 and subsequent versions. A copy of this license must be distributed with the program. It is also at: http://www.gnu.org/licenses/gpl-2.0.html.

The ordinal calculator source code, documentation and some executables can be downloaded from: http://www.mtnmath.com/ord or https://sourceforge.net/projects/ord.

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

This is a command line interactive interface to a program for exploring the ordinals. It supports recursive ordinals up to and beyond the the Bachmann-Howard ordinal[11]. It defines notations for the Church-Kleene ordinal and some larger countable ordinals. We refer to these as admissible level ordinals. They are used in a form of ordinal collapsing to define large recursive ordinals.

B.2  Command line options

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

‘cmd’ — Read specified command file and enter command line mode.
‘cmdTex’ — Read specified command file and enter TeX command line mode.
‘version’ — Print program version.
‘help’ — Describe command line options.
‘cmdDoc’ — Write manual for command line mode in TeX format.
‘tex’ — Output TeX documentation files.
'psi' — Do tests of Veblen hierarchy.
'base' — Do tests of base class Ordinal.
'try' — Do tests of class FiniteFuncOrdinal.
'gamma' — Test for consistent use of gamma and epsilon.
'iter' — Do tests of class IterFuncOrdinal.
'admis' — Do tests of class AdmisLevOrdinal.
'admis2' — Do additional tests of class AdmisLevOrdinal.
'play' — Do integrating tests.
'descend' — Test descending trees.
'collapse' — Ordinal collapsing tests.
'nested' — Ordinal nested collapsing tests.
'nested2' — Ordinal nested collapsing tests 2.
'nested3' — Ordinal nested collapsing tests 3.
'exitCode' — LimitElement exit code base test.
'exitCode2' — LimitElement exit code base test 2.
'limitEltExitCode' — Admissible level LimitElement exit code test 0.
'limitEltExitCode1' — Admissible level limitElement exit code test 1.
'limitEltExitCode2' — Admissible level limitElement exit code test 2.
'limitEltExitCode3' — Admissible level limitElement exit code test 3.
'limitOrdExitCode' — Admissible level limitOrd exit code test.
'limitOrdExitCode1' — Admissible level limitOrd exit code test.
'limitOrdExitCode2' — Admissible level limitOrd exit code test.
'limitOrdExitCode3' — Admissible level limitOrd exit code test.
'admisLimitElementExitCode' — Admissible level limitElement exit code test.
'admisDrillDownLimitExitCode' — Admissible level drillDownLimitElement exit code test.
'admisDrillDownLimitComExitCode' — Admissible level drillDownLimitElementCom exit code test.
'admisLimitElementComExitCode' — Admissible level exit code test for limitElementCom.
'admisExamples' — Admissible level tests of examples.
'nestedLimitEltExitCode' — Nested level limitElement exit code test.
'nestedLimitEltExitCode2' — Nested level limitElement exit code test 2.
'nestedBaseLimitEltExitCode' — Nested level base class limit exit code test.
'nestedLimitOrdExitCode' — Nested level limitOrd exit code test.
'nestedCmpExitCode' — Nested level compare exit code test.
'nestedEmbedNextLeast' — Nested embed next least test.
'transition' — Admissible level transition test.
'cmpExitCode' — Admissible level compare exit code test.
'drillDownExitCode' — Admissible level compare exit code test.
'embedExitCode' — Admissible level compare exit code test.
'fixedPoint' — test fixed point detection.
'nestedEmbed' — basic nested embed tests.
'infoLimitTypeExamp' — test inforLimitTypeExamples.
'cppTest' — test interactive C++ code generation.
'cppTestTeX' — test interactive C++ code that generates TeX.
'test' — a stub at the end of validate.cpp used to debug code.
B.3 Help topics

Following are topics you can get more information about by entering ‘help topic’.

‘cmds’ – lists commands.
‘defined’ – list predefined ordinal variables.
‘compare’ – describes comparison operators.
‘members’ – describes member functions.
‘ordinal’ – describes available ordinal notations.
‘ordlist’ – describes ordinal lists and their use.
‘purpose’ – describes the purpose and philosophy of this project.
‘syntax’ – describes syntax.
‘version’ – displays program version.

This program supports line editing and history.
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).

B.4 Ordinals

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

The ordinal of the integers, \( \omega \), is also represented by the single lowercase letter: ‘w’. The Veblen function is specified as \( \psi(p_1,p_2,\ldots,p_n) \) where \( n \) is any integer > 0. Special notations are displayed in some cases. Specifically \( \psi(x) \) is displayed as \( w^x \). \( \psi(1,x) \) is displayed as \( \epsilon(x) \). \( \psi(1,x,0) \) is displayed as \( \Gamma(x) \). In all cases the displayed version can be entered as input.

Larger ordinals are specified as \( \psi_{\{px\}}(p_1,p_2,\ldots,p_n) \). The first parameter is enclosed in brackets not parenthesis. \( \psi_{\{1\}} \) is defined as the union of \( w \), \( \epsilon(0) \), \( \gamma(0) \), \( \psi(1,0,0,0) \), \( \psi(1,0,0,0,0) \), \( \psi(1,0,0,0,0,0) \), \( \psi(1,0,0,0,0,0,0) \), ... You can access the sequence whose union is a specific ordinal using member functions. Type help members to learn more about this. Larger notations beyond the recursive ordinals are also available in this implementation. See the documentation ‘A Computational Approach to the Ordinal Numbers’ to learn about ‘Countable admissible ordinals’.

There are several predefined ordinals. ‘w’ and ‘\( \omega \)’ can be be used interchangeably for the ordinal of the integers. ‘eps0’ and ‘\( \omega_1CK \)’ are also predefined. Type ‘help defined’ to learn more.
B.5 Predefined ordinals

The predefined ordinal variables are:

\[
\begin{align*}
\omega & = \omega \\
\omega_1 & = \omega_1 \\
\omega_1 \text{CK} & = \omega_1 \\
\varepsilon_0 & = \varepsilon_0
\end{align*}
\]

B.6 Syntax

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

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

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

B.7 Ordinal lists

Lists are a sequence of ordinals. An assignment statement can name a single ordinal or a list of them separated by commas. In most circumstances only the first element in the list is used, but some functions (such as member function ‘limitOrdLst’) use the full list. Type ‘help members’ to learn more about ‘limitOrdLst’.
B.8 Commands

B.8.1 All commands

The following commands are available:
‘cmpCheck’ – toggle comparison checking for debugging.
‘cppList’ – list the C++ code for assignments or write to an optional file.
‘examples’ – shows examples.
‘exit’ – exits the program.
‘exportTeX’ – exports assignments statements in TeX format.
‘help’ – displays information on various topics.
‘list’ – lists assignment statements.
‘log’ – write log file (ord.log default) (see help logopt).
‘listTeX’ – lists assignment statements in TeX format.
‘logopt’ – control log file (see help log).
‘name’ – toggle assignment of names to expressions.
‘opts’ – controls display format and other options.
‘prompt’ – prompts for ENTER with optional string argument.
‘quit’ – exits the program.
‘quitf’ – exits only if not started in interactive mode.
‘read’ – read “input file” (ord_calc.ord default).
‘readall’ – same as read but no ‘wait for ENTER’ prompt.
‘save’ – saves assignment statements to a file (ord_calc.ord default).
‘setDbg’ – set debugging options.
‘tabList’ – lists assignment values as C++ code to generate a LaTeX table.
‘yydebug’ – enables parser debugging (off option).

B.8.2 Commands with options

Following are the commands with options.

Command ‘examples’ – shows examples.
It has one parameter with the following options.
‘arith’ – demonstrates ordinal arithmetic.
‘compare’ – shows compare examples.
‘display’ – shows how display options work.
‘member’ – demonstrates member functions.
‘VeblenFinite’ – demonstrates Veblen functions of a finite number of ordinals.
‘VeblenExtend’ – demonstrates Veblen functions iterated up to a recursive ordinal.
‘admissible’ – demonstrates countable admissible level ordinal notations.
‘admissibleDrillDown’ – demonstrates admissible notations dropping down one level.
‘admissibleContext’ – demonstrates admissible ordinal context parameters.
‘list’ – shows how lists work.
‘desLimitOrdLst’ – shows how to construct a list of descending trees.
Command ‘logopt’ – control log file (see help log).
It has one parameter with the following options.
‘flush’ – flush log file.
‘stop’ – stop logging.

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

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

B.9 Member functions
Every ordinal (except 0) is the union of smaller ordinals. Every limit ordinal is the union of
an infinite sequence of smaller ordinals. Member functions allow access to to these smaller
ordinals. One can specify how many elements of this sequence to display or get the value
of a specific instance of the sequence. For a limit ordinal, the sequence displayed, were it
extended to infinity and its union taken, that union would equal the original ordinal.

The syntax for a member function begins with either an ordinal name (from an assignment
statement) or an ordinal expression enclosed in parenthesis. This is followed by a dot (.) and
then the member function name and its parameters enclosed in parenthesis. The format is
‘ordinal_name.memberFunction(p)’ where p may be optional. Functions ‘limitOrdLst’ and
‘desLimitOrdLst’ return a list. All other member functions return a scalar value. Unless
specified otherwise, the returned value is that of the ordinal the function was called from.
The member functions are:
‘cpp’ – output C++ code to define this ordinal.
‘descend’ – (n,m) iteratively (up to m) take nth limit element.
‘descendFull’ – (n,m,k) iteratively (up to m) take n limit elements with root k.
‘desLimitOrdLst’ – (depth, list) does limitOrdLst iteratively on all outputs depth times.
‘ek’ – effective kappa (or indexCK) for debugging.
‘getCompareIx’ – display admissible compare index.
‘isValidLimitOrdParam’ – return true or false.
‘iv’ – alias for isValidLimitOrdParam.
‘le’ – evaluates to specified finite limit element.
‘lec’ – alias to return limitExitCode (for debugging).
‘limitElement’ – an alias for ‘le’.
‘limitExitCode’ – return limitExitCode (for debugging).
‘listLimitElts’ – lists specified (default 10) limit elements.
‘listElts’ – alias for listLimitElts.
‘limitOrd’ – evaluates to specified (may be infinite) limit element.
‘limitOrdLst’ – apply each input from list to limitOrd and return that list.
‘lo’ – alias for limitOrd.
‘limitType’ – return limitType.
‘maxLimitType’ – return maxLimitType.
‘maxParameter’ – return maxParameter (for debugging).

B.10 Comparison operators

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

B.11 Examples

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

To select an examples type ‘examples’ followed by one of the following options.
‘arith’ – demonstrates ordinal arithmetic.
‘compare’ – shows compare examples.
‘display’ – shows how display options work.
‘member’ – demonstrates member functions.
‘VeblenFinite’ – demonstrates Veblen functions of a finite number of ordinals.
‘VeblenExtend’ – demonstrates Veblen functions iterated up to a recursive ordinal.
‘admissible’ – demonstrates countable admissible level ordinal notations.
‘admissibleDrillDown’ – demonstrates admissible notations dropping down one level.
‘admissibleContext’ – demonstrates admissible ordinal context parameters.
‘list’ – shows how lists work.
‘desLimitOrdLst’ – shows how to construct a list of descending trees.
B.11.1 Simple ordinal arithmetic

The following demonstrates ordinal arithmetic.

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

B.11.2 Comparison operators

The following shows compare examples.

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

B.11.3 Display options

The following shows how display options work.

```
ordCalc> a=w^(w^w)
Assigning ( w^( w^w ) ) to ‘a’.
ordCalc> b=epsilon(a)
Assigning epsilon( ( w^( w^w ) ) ) to ‘b’.
ordCalc> c=gamma(b)
Assigning gamma( epsilon( ( w^( w^w ) ) ) ) to ‘c’.
ordCalc> list
a = ( w^( w^w ) )
b = epsilon( ( w^( w^w ) ) )
c = gamma( epsilon( ( w^( w^w ) ) ) )
%Total 3 variables listed.
ordCalc> opts tex
ordCalc> list
a = \omega{}^{\omega{}^{\omega{}}}
```
\[ b = \varepsilon \]
\[ c = \Gamma \]
\%Total 3 variables listed.

B.11.4 Member functions

The following demonstrates member functions.

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

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

Normal form: psi( 1, 0, 0, 0, 0 )
ordCalc> b=a.le(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’.

B.11.5 Veblen function of N ordinals

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

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

ordCalc> a=psi(w,w)
Assigning psi( w, w ) to ‘a’.
ordCalc> b=psi(a,3,1)
Assigning $\psi(\psi(w,w),3,1)$ to 'b'.

ordCalc> b.listElts(3)

3 limitElements for $\psi(\psi(w,w),3,1)$

le(1) = $\psi(\psi(w,w),3,0)$

le(2) = $\psi(\psi(w,w),2,\psi(\psi(w,w),3,0)+1)$

le(3) = $\psi(\psi(w,w),2,\psi(\psi(w,w),2,\psi(\psi(w,w),3,0)+1)+1)$

End limitElements

Normal form: $\psi(\psi(w,w),3,1)$

ordCalc> c=\psi\{a\}(a,a,b,1,3)

Assigning $\psi\{a\}(a,a,b,1,3)$ to 'c'.

ordCalc> c.listElts(3)

3 limitElements for $\psi(\psi(w,w),\psi(w,w),\psi(\psi(w,w),3,1),1,3)$

le(1) = $\psi(\psi(w,w),\psi(w,w),\psi(\psi(w,w),3,1),1,2)$

le(2) = $\psi(\psi(w,w),\psi(w,w),\psi(\psi(w,w),3,1),0,\psi(\psi(w,w),3,1),1,2)+1)$

le(3) = $\psi(\psi(w,w),\psi(w,w),\psi(\psi(w,w),3,1),0,\psi(\psi(w,w),3,1),0,\psi(\psi(w,w),\psi(w,w),\psi(\psi(w,w),3,1),1,2)+1)+1)$

End limitElements

Normal form: $\psi(\psi(w,w),\psi(w,w),\psi(\psi(w,w),3,1),1,3)$

### B.11.6 Extended Veblen function

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

The extended Veblen function, $\psi\{a\}(x_1,x_2,...,x_n)$, iterates the idea of the Veblen function up to any recursive ordinal. The first parameter is the recursive ordinal of this iteration.

ordCalc> a=psi\{1\}(1)

Assigning $\psi\{1\}(1)$ to 'a'.

ordCalc> a.listElts(4)

4 limitElements for $\psi\{1\}(1)$

le(1) = $\psi\{1\} + 1$

le(2) = $\psi(\psi\{1\} + 1,0)$

le(3) = $\psi(\psi\{1\} + 1,0,0)$

le(4) = $\psi(\psi\{1\} + 1,0,0,0)$

End limitElements
Normal form: $\psi_{w+1}(3)$

Assigning $\psi_{w+1}(3)$ to ‘b’.

$\text{ordCalc} > \text{b.listElts}(4)$

4 limitElements for $\psi_{w+1}(3)$

le(1) = $\psi_{w+1}(2) + 1$
le(2) = $\psi_w(\psi_{w+1}(2) + 1, 0)$
le(3) = $\psi_w(\psi_{w+1}(2) + 1, 0, 0)$
le(4) = $\psi_w(\psi_{w+1}(2) + 1, 0, 0, 0)$

End limitElements

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

**B.11.7 Admissible countable ordinal notations**

The following demonstrates countable admissible level ordinal notations.

Countable admissible level notations, $\omega_{k,g}(x_1,x_3,..,x_n)$ extend the idea of recursive notation to larger countable ordinals. The first parameter is the admissible level. $\omega_{1}$ is the Church-Kleene ordinal. The remaining parameters are similar to those defined for smaller ordinal notations.

$\text{ordCalc} > a=w_{1}(1)$

Assigning $\omega_{1}(1)$ to ‘a’.

$\text{ordCalc} > a.listElts(4)$

4 limitElements for $\omega_{1}(1)$

le(1) = $\omega_{1}$
le(2) = $\psi_{\omega_{1} + 1}$
le(3) = $\psi_{\psi_{\omega_{1} + 1} + 1}$
le(4) = $\psi_{\psi_{\psi_{\omega_{1} + 1} + 1} + 1}$

End limitElements

Normal form: $\omega_{1}(1)$

**B.11.8 Admissible notations drop down parameter**

The following demonstrates admissible notations dropping down one level.

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

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

4 limitElements for omega._{ 1}[ 1]
le(1) = w
le(2) = psi._{ w}
le(3) = psi._{ psi._{ w} + 1}
le(4) = psi._{ psi._{ psi._{ w} + 1} + 1}
End limitElements

Normal form: omega._{ 1}[ 1]
```

```
ordCalc> b=w_[1]
Assigning omega._{ 1} to 'b'.
ordCalc> c=b.limitOrd(w^3)
Assigning omega._{ 1}[ ( w^3 )] to 'c'.
ordCalc> c.listElts(4)

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

Normal form: omega._{ 1}[ ( w^3 )]
```

```
ordCalc> d=w_[5,c](3,0)
Assigning omega._{ 5, omega._{ 1}[ ( w^3 )]}(3, 0) to 'd'.
ordCalc> d.listElts(4)

4 limitElements for omega._{ 5, omega._{ 1}[ ( w^3 )]}(3, 0)
le(1) = omega._{ 5, omega._{ 1}[ ( w^3 )]}(2, 1)
le(2) = omega._{ 5, omega._{ 1}[ ( w^3 )]}(2, omega._{ 5, omega._{ 1}[ ( w^3 )]}(2, 1) + 1) + 1)
le(3) = omega._{ 5, omega._{ 1}[ ( w^3 )]}(2, omega._{ 5, omega._{ 1}[ ( w^3 )]}(2, omega._{ 5, omega._{ 1}[ ( w^3 )]}(2, omega._{ 5, omega._{ 1}[ ( w^3 )]}(2, 1) + 1) + 1)
End limitElements

Normal form: omega._{ 5, omega._{ 1}[ ( w^3 )]}(3, 0)
```
B.11.9 Admissible notations context parameter

The following demonstrates admissible ordinal context parameters.

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

\[
\text{ordCalc} > a=[[1]]\omega_{\{1\}}
\]
Assigning \([[1]]\omega_{\{1\}}\) to ‘a’.

\[
\text{ordCalc} > a.\text{listElts}(4)
\]

4 limitElements for \([[1]]\omega_{\{1\}}\)

\[
\begin{align*}
\text{le}(1) &= [[1]]\omega_{\{1\}}[[ 1]] \\
\text{le}(2) &= [[1]]\omega_{\{1\}}[[ 2]] \\
\text{le}(3) &= [[1]]\omega_{\{1\}}[[ 3]] \\
\text{le}(4) &= [[1]]\omega_{\{1\}}[[ 4]]
\end{align*}
\]
End limitElements

Normal form: \([[1]]\omega_{\{1\}}\)

B.11.10 Lists of ordinals

The following shows how lists work.

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

\[
\text{ordCalc} > \text{lst} = 1, 12, w, \text{gamma}(w^w), w1
\]
Assigning 1, 12, w, \text{gamma}(w^w), \omega_{\{1\}} to ‘lst’.

\[
\text{ordCalc} > a=w.1.\text{limitOrdLst}(\text{lst})
\]

\[
\begin{align*}
\text{omega}_{\{1\}}(12) &= \omega_{\{1\}}[12] \\
\text{omega}_{\{1\}}(w) &= \omega_{\{1\}}[w] \\
\text{omega}_{\{1\}}(\text{gamma}(w^w)) &= \omega_{\{1\}}[\text{gamma}(w^w)]
\end{align*}
\]
Assigning \omega_{\{1\}}[12], \omega_{\{1\}}[w], \omega_{\{1\}}[\text{gamma}(w^w)] to ‘a’.

\[
\text{ordCalc} > \text{bg} = w_{[w+33]}
\]
Assigning \omega_{[w+33]} to ‘bg’.

\[
\text{ordCalc} > c=\text{bg}.\text{limitOrdLst}(\text{lst})
\]

\[
\begin{align*}
\text{omega}_{\{w+33\}}(12) &= \omega_{\{w+33\}}[12] \\
\text{omega}_{\{w+33\}}(w) &= \omega_{\{w+33\}}[w] \\
\text{omega}_{\{w+33\}}(\text{gamma}(w^w)) &= \omega_{\{w+33\}}[\text{gamma}(w^w)] \\
\text{omega}_{\{w+33\}}(\omega_{\{1\}}) &= \omega_{\{w+33\}}[\omega_{\{1\}}]
\end{align*}
\]
Assigning \omega_{\{w+33\}}[12], \omega_{\{w+33\}}[w], \omega_{\{w+33\}}[\text{gamma}(w^w)] to ‘c’.
\[ w^w \] \[ \omega \{ w + 3 \} \[ \omega \{ 1 \} \] to 'c'.

### B.11.11 List of descending trees

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

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

```
ordCalc> lst = 1, 5, w, psi(2,3)
Assigning 1, 5, w, psi(2,3) to 'lst'.
ordCalc> bg = w{3}
Assigning \omega{3} to 'bg'.
ordCalc> d = bg.desLimitOrdLst(2, lst)
( \omega{3}[1] ).limitOrd(1) = \omega{2}
( \omega{3}[1] ).limitOrd(5) = \omega{2, \omega{2, \omega{2, \omega{2, \omega{w} + 1} + 1} + 1} + 1}
( \omega{3}[5] ).limitOrd(1) = \omega{3}[4]
( \omega{3}[5] ).limitOrd(5) = \omega{2, \omega{2, \omega{2, \omega{2, \omega{w} + 1} + 1} + 1} + 1}
( \omega{3}[w] ).limitOrd(1) = \omega{3}[1]
( \omega{3}[w] ).limitOrd(5) = \omega{3}[5]
( \omega{3}[psi(2,3)] ).limitOrd(1) = \omega{3}[psi(2,2)]
( \omega{3}[psi(2,3)] ).limitOrd(5) = \omega{3}[epsilon(epsilon(epsilon(epsilon(psi(2,2) + 1) + 1) + 1) + 1)]
```

Assigning \omega{3}[1], \omega{3}[5], \omega{3}[w], \omega{3}[psi(2,3)], \omega{2}, \omega{2, \omega{2, \omega{2, \omega{2, \omega{w} + 1} + 1} + 1} + 1}], \omega{2, \omega{2, \omega{2, \omega{2, \omega{w} + 1} + 1} + 1} + 1}], \omega{3}[1], \omega{3}[5], \omega{3}[psi(2,2)], \omega{3}[epsilon(epsilon(epsilon(epsilon(psi(2,2) + 1) + 1) + 1) + 1)] to 'd'.
References


[3] Paul Budnik. What is and what will be: Integrating spirituality and science. Mountain Math Software, Los Gatos, CA, 2006. 6, 8, 102, 109, 119


Index

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

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

Symbols

< 125  
<= 125  
== 125  
> 125  
>= 125  
\(\Delta\) 19, 21  
\(\Gamma\) 25  
\(\Omega\) 10, 49  
\(\Psi(\alpha)\) 49  
\(\Psi(\varepsilon_{\Omega+1})\) 50  
\(\delta_{ck}\) 68  
\(\emptyset\) 112  
\(\exists\) 112  
\(\exists!\) 113  
\(\forall\) 112  
\(\omega\) 7, 9, 13, 19, 44, 109, 113  
\(\omega_{1}\) 8, 44, 56  
\(\omega_{1}\) limitOrd 47  
\(\omega_{1}\) limitOrd 47  
\(\omega_{2}\) 44  
\(\subseteq\) 114  
\(\varepsilon_{0}\) 9, 12, 19, 22  
\(\varepsilon_{\alpha}\) 25  
\(\varphi(1,0)\) 12  
\(\varphi(\alpha,\beta)\) 19  
\(\varphi(1,0)\) 22  
\(\varphi_{1}\) 24  
\(\varphi_{\beta}\) 24  
\(\{x\}\) 113  
& 25  
\(=\) 112  
\([n]\) 48  
\(*\) 13  
\(\wedge\) 13, 15  
\(*\) 121  
\(\wedge\) 121  

FiniteFuncOrdinal operators 31  
+ 13, 121

A

AA 29  
AB 29, 40  
Addition 16  
addition, FiniteFuncOrdinal 31  
Ordinal 16  
admisCodeLevel 60  
admisCodeLevel 58, 62, 69, 77, 79  
admisLevelFunctional 56  
C++ examples 58  
AdmisLevOrdinal 45, 57, 88  
AdmisLevOrdinal class 56  
AdmisLevOrdinal operators 79  
AdmisLevOrdinal::fixedPoint 77  
AdmisLevOrdinal::limitElement examples 70, 71, 72  
AdmisLevOrdinal::limitOrd examples 78  
AdmisNormalElement 45, 56  
AdmisNormalElement::compare 56, 58, 60, 88  
AdmisNormalElement::compareCom 59, 60  
AdmisNormalElement::drillDownLimitElement 65  
exit codes 67  
AdmisNormalElement::drillDownLimitElementCom 65  
exit codes 68  
AdmisNormalElement::Embeddings::  
AdmisNormalElement::isValidLimitOrdParam 73, 102  
AdmisNormalElement::increasingLimit 74, 93  
AdmisNormalElement::isLimitInfo 73  
exit codes 66  
AdmisNormalElement::limitElementCom 64  
exit codes 67  
AdmisNormalElement::limitInfo 73  
AdmisNormalElement::limitOrd 73  
AdmisNormalElement::limitOrdCom 75  
exit codes 76  
AdmisNormalElement::maxLimitType 73, 102  
AdmisNormalElement::paramLimitElement 97  
cases 97  
admissible countable ordinal notations example 129  
index 44, 45, 56  
level ordinal 44
Admissible level ordinal collapsing 53
admissible level ordinal notations 45
notations context parameter example 131
notations drop down parameter example 129
ordinal 44
admissible 123, 125
admissibleContext 123, 125
admissibleDrillDown 123, 125
all 124
arith 123, 124, 125
assignment statement 122
axiom of choice 114
of extensionality 112
of infinity 113
of power set 114
of replacement, objective 116
of the empty set 112
of union 112
of unordered pairs 112
scheme of replacement 113
axioms of ZFC 111, 114

B
Bachmann-Howard ordinal 50
barber paradox 113
base class 12
beyond recursive ordinals 10
both 124

C
C 87
calculator command options 123
commands 123
examples 125
list 131
ordinal 6
syntax 122
Cantor diagonalization 105
Georg 102
normal form 12
Cantor’s proof 102
cantorCodeLevel 31, 32, 34
CantorNormalElement 13, 14, 24, 32, 34
CantorNormalElement::compare 14, 25
CantorNormalElement::embedType 73, 102
CantorNormalElement::expTerm 34
CantorNormalElement::getMaxParameter 28
CantorNormalElement::limitElement 15, 28
cfp 59, 60
choice, axiom of 114
Church, Alonzo 8, 44
Church-Kleene 80
ordinal 8, 44, 56, 129
class 11, 12
constructor 13
class NestedEmbeddings 91
clear 124
cmds 121
cmp 59
cmpCheck 123
code, exit 28, 38, 59, 63
source 6
codeLevel 17, 25, 31, 61, 79
collapsing, admissible level ordinal 53
function 49
function, ordinal 51
ordinal 49
command line options 119
commands, calculator 123
with options 123
compare member function 88
cmpareCom 59, 60
compareFiniteParams 59
comparison operators 125
operators example 126
completed infinite 102
computational interpretation 10
computer halting problem 10
construct 124
constructor 28
class 13
continuous function 19
continuum hypothesis 11, 106
cosmology 106
countable 102
admissible ordinal 44, 56
cpp 124
cppList 56, 57, 88, 89, 123
createParameters 25
createVirtualOrd 61
createVirtualOrdImpl 61
creative philosophy 117
cut, Dedekind 11

D
DCAL 65, 67, 68, 87, 98, 101
DCBO 65, 68, 87, 98, 101
DCCS 65, 67, 68, 87, 98, 101
DCDO 65, 67, 68, 87
DCES 65, 68, 87
dd 59
of admissible notations context parameter

of admissible notations drop down parameter

of comparison operators

of display options

of extended Veblen function

of list of descending trees

of lists of ordinals

of member functions

of simple ordinal arithmetic

of Veblen function of N ordinals

easy examples

AdmisLevOrdinal::limitElement

AdmisLevOrdinal::limitOrd

FiniteFuncOrdinal C++ code

finiteFunctional C++ code

FiniteFuncOrdinal exponential

finiteFunctional Ordinal Calculator code

_iterativeFunctional C++ code

IterFuncOrdinal exponential

IterFuncOrdinal multiply

NestedEmbedOrdinal interpreter code

NestedEmbedOrdinal::drillDownLimitElementEq

finiteFunctional::embedLimitElement

NestedEmbedOrdinal::limitElement

NestedEmbedOrdinal::limitOrd

Ordinary exponential

Ordinal exponentiation C++ code

Ordinal multiply

Ordinal::limitElement

Ordinal::limitOrd

Ordinal::multiply

examples
easy examples

examples of simple ordinal functions

example of simple ordinal functions

example of simple ordinal functions

example of simple ordinal functions

example of simple ordinal functions

example of simple ordinal functions

example of simple ordinal functions

example of simple ordinal functions

example of simple ordinal functions

example of admissible countable ordinal notations

example of admissible countable ordinal notations

example of admissible countable ordinal notations


exp 124
explicit incompleteness 10 102
exponentiation 17
Ordinal 17
exportTeX 123
extended Veblen function example 128
extensionality, axiom of 112

F

factor 79
FALSE 125
FB 29, 30, 40, 87
FD 29, 30, 40, 87
Feferman, Solomon 106
finite function hierarchy 21
    function normal form 22
    functions limitElement 22
finiteFuncCodeLevel 24, 28, 32
FiniteFuncNormalElement 24
FiniteFuncNormalElement::compare 25, 37
    FiniteFuncNormalElement::
        compareFiniteParams 28
        FiniteFuncNormalElement::doMultiply 32
        FiniteFuncNormalElement::doToPower 34
        FiniteFuncNormalElement::
            limitElement 28
        FiniteFuncNormalElement::
            limitElementCom 29
    FiniteFuncNormalElement::
        limitElementCom exit codes 30
    FiniteFuncNormalElement::limitOrd 36
FiniteFuncNormalElement::
    limitOrdCom 36
FiniteFuncOrdinal 25, 36
    class 24
    addition 31
    C++ examples 27
    constructor 25
    exponential examples 35
    exponentiation 34
    multiplication 31
    multiply examples 33
    FiniteFuncOrdinal::compare 25
    FiniteFuncOrdinal::fixedPoint 31
    FiniteFuncOrdinal::limitElement 28
finiteFunctional 24, 26, 27, 31
    C++ examples 27
    Ordinal Calculator examples 26
FiniteFunctionNormalElement::
    limitType 73 102
finiteLimit 62 86
    FiniteFuncNormalElement::limitOrd 36

fixed point 19, 25, 34, 36
    points 9
fixedPoint 31 41, 77, 102
FL 29, 30, 40, 87, 98
flush 124
FN 29, 30, 40, 87
FOB 76
FOC 76
form, normal 14, 45, 56
formalization objective parts of ZF 116
funcParameters 25
    function, collapsing 49
        continuous 19
        normal 19
        Veblen 121
    functionLevel 59
    functionNxtSucc 39, 62, 69, 84, 86, 87, 94
    functions, member 11 124
        ordinal 9
    functionSucc 39, 62, 69, 82, 86, 87, 94

G
gamma(x) 121
getCompareIx 125
getMaxParameter() 59, 91
GNU General Public License 6
Gnu Multiple Precision Arithmetic 19
God Created the Integers 11

H
help 123
hierarchy, ordinal 109
    Veblen 9, 11, 12, 19
Howard, Bachmann- ordinal 50
hypothesis, continuum 11 106

I

ideal, Platonic 102
IF 39, 40
IfNe 63
IG 39, 40, 87
ignf 59, 60
ignoreFactor 59
II 39, 40, 67, 87
IJ 39, 40, 87
IK 39, 40, 87, 98
impredicative 10, 102
incompleteness, explicit 10, 102
Kleene, Stephen C. 8 44

Löwenheim, Leopold 105
Löwenheim-Skolem theorem 105
LCAF 64 67
LCBL 64, 66, 67, 87
LCCI 64 67
LCDP 64, 66, 67, 87, 100
LCEL 64, 66, 67, 87, 98
le 63, 96, 125
LEAD 64, 66, 67, 68, 101
leastIndexLimit 82, 83, 86, 87, 94, 96, 103, 105
leastIndexLimitParam 82, 86, 87, 94, 96, 103, 105
leastIndexSuccParam 84, 86, 87, 94, 97
leastLevelLimit 81, 86, 87, 94, 96, 103, 105
leastLevelLimitParam 84, 86, 87, 94, 96, 103, 105
leastLevelSuccParam 84, 86, 87, 94, 97
LEBC 64, 66, 67
le 46, 125
LECK 64, 66, 87
LEDC 64, 66, 87
LEDE 64, 66, 87
LEEE 64, 66, 87
leUse 61, 63, 74
levelDecr 95
levelDecrIndexSet 95
limArith 124
limit 124
limitElement 14, 28, 38, 46, 61, 91, 125
exit code 23
finite functions 22
exitElementCom 64, 94
limitExitCode 46, 125
limitInfo 73, 102
LimitInfoType 74, 102
limitOrd 36, 46, 56, 73, 102, 125, 132
limitOrdA 81
limitOrdCom 61
limitOrdLst 125, 131
limitType 46, 47, 56, 73, 102, 125
of admissible level notations 49
list, calculator 131
of descending level notations 132
list 123, 125
listElts 125
listLimitElts 125
lists of ordinals example 131
    ordinal 122
listTeX 123
lo 63, 125
LOA 75, 76, 104
loa 96
LOB 75, 76, 104
LOC 75, 76
LOD 75, 76, 104
LOE 75, 76
LOF 75, 76, 104
log 123
logopt 123, 124
looking glass reality 10
lp1 63
lsG 63
lsx 63

M
Machines, Turing 10
management, memory 13
Mandelbrot 53, 80
    set 47
manual, user’s 6
material, tutorial 6
Mathematica, Principia 47
mathematical objects 110
mathematics’ inherent incompleteness 105
maxLimitType 46, 47, 56, 73, 102, 125
maxParameter 59, 125
maxParamFirstTerm 59
member functions 11, 124
    functions example 127
member 123, 125
members 121
memory management 13
Mpir 13
multiple precision arithmetic 13
multiplication 16
    Ordinal 16
multiply 31, 32
multiplyBy 17, 31, 32

N
name 123
namespace ord 127
NEA 96, 100
NEB 87, 96, 100
NEC 87, 96, 100
NED 96, 98, 100
NEF 87, 96, 98, 100
NEG 87, 96, 100
NEH 87, 96, 100
nested examples 55
nestedEmbedCodeLevel 92
nestedEmbedCodeLevel 86, 87, 88, 105
NestedEmbeddings 88, 90, 91
NestedEmbeddings::compare 91
NestedEmbeddings::nextLeast 91
nestedEmbedFunctional 88
NestedEmbedNormalElement::
    limitElement 91
NestedEmbedNormalElement transitions 101
NestedEmbedNormalElement::compare 88, 92
NestedEmbedNormalElement::
    drillDownLimitElementEq 95, 99
    examples 99
NestedEmbedNormalElement::
    embedLimitElement 96, 100
    examples 100
NestedEmbedNormalElement::
    limitElement 94, 98
    examples 98
NestedEmbedNormalElement::limitInfo 102
NestedEmbedNormalElement::limitOrd 102, 103, 104
    examples 104
NestedEmbedNormalElement::
    limitOrdCom 102
NestedEmbedNormalElement::
    NestedEmbeddings::
        compare 91
NestedEmbedNormalElement::
    paramLimitElement examples 100
NestedEmbedOrdinal 88, 89
NestedEmbedOrdinal class 88
NestedEmbedOrdinal interpreter code examples 90
NestedEmbedOrdinal::fixedPoint 102
nestEmbed 92
nextLeast 91, 95, 97
NLA 94, 98, 100
NLB 94, 95, 98, 101
NLC 94, 98, 101
NLE 94, 98
NLP 94, 98, 100
nlSg 63
nlSx 63
NOA 103
NOB 103, 104
NOC 103, 104
NOD 103
NOE 103
NOF 103

140
normal form 14, 45, 56
form, finite function 22
function 19
normalForm 14
not Ψ displayable 53
notation, ordinal 7, 9
recursive ordinal 109
nullLimitType 46, 47

O

object oriented language 11
objective axiom of replacement 116
interpretation ZF 117
parts of ZF 115
parts of ZF, formalization 116
objects, mathematical 110
OFA 36, 76
OFB 36, 76, 104
omega 121, 122
omega w 121
omegaCK 122
Operators 15
operators, comparison 125
opts 123, 124
oracle 10
TM 44
ord namespace 12
ordinal 7, 121
admissible 44
admissible level 44
arithmetic examples 16
Bachmann-Howard 50
calculator 6
Church-Kleene 8, 44, 56, 129
collapsing 49
collapsing function 51
functions 9
hierarchy 109
induction 8
limit 7
lists 122
normal form 12
notation 7, 9
recursive 6, 7, 10, 19, 44
successor 7
uncountable 10

Ordinal 10, 11, 12, 56, 121
addition 16
base class 13
exponential examples 20
exponentiation 17
expressions 13
multiplication 16
multiply examples 18
operators 15
Ordinal::compare 14, 25
Ordinal::isValidLimitOrdParam 46
Ordinal::limitElement 14, 22, 61, 93
examples 15
Ordinal::limitOrd 46
Ordinal::limitType 46
Ordinal::maxLimitType 46
Ordinal::normalForm 14
Ordinal::psiNormalForm 53
Ordinal::texNormalForm 14
OrdinalImpl 28
OrdinalImpl::embedType 46
OrdinalImpl::limitElement 28
ordinals, infinite 106
predefined 122
recursive 109
ordlist 121

P

paradox, barber 113
parameterCompare 59
parameters, typed 46
paramLimitElement 94, 97
examples 100
paramsSucc 29, 39, 62, 69, 84, 86, 87, 94
collapsing 49
collapsing function 51
functions 9
hierarchy 109
induction 8
limit 7
lists 122
normal form 12
notation 7, 9
recursive 6, 7, 10, 19, 44
successor 7
uncountable 10

Principia Mathematica 47
proof, Cantor’s 102
properties of properties 111
relative 111
psi px(p1,p2,...,pn) 121
psi 24, 121, 124
psi(1,x) 121
psi(1,x,0) 121
psi(x) 121
psiNormalForm 53
pseudoCodeLevel 31, 41
purpose 121

Q
quantification over the reals 10
quantifier, existential 112
universal 112
quit 123
quitf 123

R
read 123
readall 123
reality, physical 106
recordLimitType 47
recursive 10
ordinary 6, 7, 10, 19, 44
ordinal notation 109
ordinals 109
recursively enumerable 10
relative properties 111
rep1 63
rep2 63
replace1 63
replace2 63
replacement axiom, objective 116
axiom scheme 113
Rtn x 63
Russell, Bertrand 47

S
Sacks, Gerald 44
save 123
set, Mandelbrot 47
setDbg 123, 124
setDbg compare 38
setDbg limit 38
sets, infinite 7
significance, physical 106
simple ordinal arithmetic example 126
Skolem, Thoralf 105
source code 6

T
tabList 123
termEmbed 92
tex 124
texNormalForm 14
text 124
theorem, incompleteness 102
this 37, 59, 63
TM 10
oracle 44
termEffEbd 92
termIxCK 92
termPmRst 92
TRUE 125
Turing Machines 10
Machines with oracles 44
tutorial material 6
typed parameters 46

U
uncountable ordinal 10
union, axiom of 112
universal quantifier 112
unknownLimit 62, 86
unordered pairs axiom 112
user’s manual 6

V
Veblen function 121
function of N ordinals example 127
hierarchy 9, 11, 12, 19
VeblenExtend 123, 125
VeblenFinite 123, 125
version 121
virtual function 11, 12

W
well founded 10, 44
ordered ordinals \text{ordinals} \text{7} \\
w^{x \text{121}} \\
 w \text{121}, \text{122} \\
 w1 \text{122} \\
w1CK \text{122} \\
 Whitehead, Alfred North \text{47}

\text{X} \\
 X \text{59}

\text{Y} \\
 yydebug \text{123}

\text{Z} \\
 Zermelo Frankel set theory \text{6, 7, 111} \\
 zeroLimit \text{62, 86} \\
 ZF \text{7, 108} \\
 \quad \text{objective interpretation 117} \\
 \quad \text{objective parts 115} \\
 \quad \text{objective parts, formalization 116} \\
 ZFC \text{111, 114} \\
 \quad \text{axioms 111, 114} \\

\text{Formatted: January 20, 2011}