Hydra Battles in Coq

Pierre Castéran
LaBRI, Univ. Bordeaux, CNRS UMR 5800, Bordeaux-INP
with contributions by Évelyne Contejean, Florian Hatat and Pascal Manoury

November 4, 2020

"I start from one point and go as far as possible."
John Coltrane.

Contents

Chapter 1

Introduction

Proof assistants are excellent tools for exploring the structure of mathematical proofs, studying which hypotheses are really needed, and which proof patterns are useful and/or necessary. Since the development of a theory is represented as a bunch of computer files, everyone is able to read the proofs with an arbitrary level of detail, or to play with the theory by writing alternate proofs or definitions.

Among all the theorems proved with the help of proof assistants like Coq, Isabelle, HOL, etc., several statements and proofs share some interesting features:

- Their statements are easy to understand, even by non-mathematicians
- Their proof requires some non-trivial mathematical tools
- Their mechanization on computer presents some methodological interest.

This is obviously the case of the four-color theorem [Gon08] and the Kepler conjecture [H$^+$15]. We do not mention impressive works like the proof of the odd-order theorem [GAA$^+$13], since understanding its statement requires a quite good mathematical culture.

Hydra games (a.k.a. *Hydra battles*) appear in an article published in 1982 by two mathematicians: L. Kirby and J. Paris [KP82]: *Accessible Independence Results for Peano Arithmetic*. Although the mathematical contents of this paper are quite advanced, the rules of hydra battles are very easy to understand. There are now several sites on Internet where you can find tutorials on hydra games, together with simulators you can play with. See, for instance, the page written by Andrej Bauer [Bau08].

Hydra battles, as well as Goodstein Sequences [Goo44, KP82] are a nice way to present complex termination problems. The article by Kirby and Paris presents a proof of termination based on ordinal numbers, as well as a proof that this termination is not provable in Peano arithmetic. In the book dedicated to J.P. Jouannaud [CLKK07], N. Dershowitz and G. Moser give a thorough survey on this topic [DM07].

Here, we present a development for the Coq proof assistant, after the work of Kirby and Paris. This formalization contains the following main parts:

- Representation in Coq of hydras and hydra battles
- A proof that every battle is finite and won by Hercules. This proof is based on a *variant* which maps any hydra to an ordinal strictly less than ϵ_0 and is strictly decreasing along any battle.
- Using a combinatorial toolkit designed by J. Ketonen and R. Solovay [KS81], we prove that, for any ordinal $\mu < \epsilon_0$, there exists no such variant mapping any hydra to an ordinal stricly less than μ. Thus, the complexity of ϵ_0 is really needed in the previous proof.
- We prove a relation between the length of a "classic" class of battles [1] and the Wainer-Hardy hierarchy of "rapidly growing functions" H_α [Wai70]. The considered class of battles, which we call *standard* is the most considered one in the scientific litterature(including popularization).

Simply put, this document tries to combines the scientific interest of two articles [KP82, KS81] and a book [Sch77] with the playful activity of truly proving theorems. We hope that such a work, besides exploring a nice piece of discrete maths, will show how Coq and its standard library are well fitted to help us to understand some non-trivial mathematical developments, and also to experiment the constructive parts of the proof through functional programming.

We also hope to provide a little clarification on infinity (both potential and actual) through the notions of function, computation, limit, types and proofs.

1.1 Remarks

1.1.1 Difference from Kirby and Paris's Work

In [KP82], Kirby and Paris showed that there is no proof of termination of all hydra battles in Peano Arithmetic (PA). Since we are used to writing proofs in higher order logic, the restriction to PA was quite unnatural for us. So we chosed to prove another statement without any reference to PA, by considering a class of proofs indexed by ordinal numbers upto ϵ_0.

1.1.2 Trust in our Proofs?

Unlike mathematical literature, where definitions and proofs are spread over many articles and books, the whole proof is now inside your computer. It is composed of the `.v` files you downloaded and parts of Coq's standard library. Thus, there is no ambiguity in our definitions and the premises of the theorems. Furthermore, you will be able to navigate through the development, using your favourite text editor or IDE, and some commands like `Search`, `Locate`, etc.

1.1.3 Assumed Redundancy

It may often happen that several definitions of a given concept, or several proofs of a given theorem are possible. If all the versions present some interest, we will make them available, since each one may be of some methodological interest (by

[1] This class is also called *standard* in this document (text and proofs). The *replication factor* of the hydra is exactly i at the i-th round of the battle (see Sect 2.0.1 on page 15).

illustrating some tactic of proof pattern, for instance). We may use Coq's module system to make several proofs of a given theorem co-exist in our libraries (see also Sect 1.1.10 on page 12). After some discussions of the pros and cons of each solution, we develop only one of them, leaving the others as exercises or projects (i.e., big or difficult exercises). In order to discuss which assumptions are really needed for proving a theorem, we will also present several aborted proofs. Of course, do not hesitate to contribute nice proofs or alternative definitions !

It may also happen that some proof looks to be useless, because the proven theorem is a trivial consequence of another (proven too) result. For instance, let us consider the three following statements:

1. There is no measure into $\mathbb{N}$ for proving the termination of all hydra battles (Sect 2.4.3 on page 39).

2. There is no measure into $[0, \omega^2)$ for proving the termination of all hydra battle (Sect 3.7.3 on page 55).

3. There is no measure into $[0, \mu)$ for proving the termination of all hydra battles, for any $\mu < \epsilon_0$ (Sect 5.4.1 on page 97).

Obviously, the third theorem implies the second one, which implies the first one. So, theoretically, a library would contain only a proof of (3) and remarks for (2) and (1). But we found it interesting to make all the three proofs available, allowing the reader to compare their common structure and notice their technical differences. In particular, the proof of (3) uses several non-trivial combinatorial properties of ordinal numbers up to ϵ_0 [KS81], whilst (1) and (2) use simple properties of $\mathbb{N}$ and $\mathbb{N}^2$.

1.1.4 About Logic

Most of the proofs we present are *constructive*. Whenever possible, we provide the user with an associated function, which she or he can apply in Gallina or OCaml in order to get a "concrete" feeling of the meaning of the considered theorem. For instance, in Chapter 5 on page 89, the notion of *limit ordinal* is made more "concrete" thanks to a function `canon` which computes every item of a sequence which converges on a given limit ordinal α. This simply typed function allows the user/reader to make her/his own experimentations. For instance, one can very easily compute the 42-nd item of a sequence which converges towards ω^{ω^ω}.

Except in the `Schutte` library, dedicated to an axiomatic presentation of the set of countable ordinal numbers, all our development is axiom-free, and respects the rules of intuitionistic logic. Note that we also use the `Equations` plug-in [SM19] in the definitition of several rapidly growing hierarchy of functions, in Chap. 6. This plug-in imports several known-as-harmless axioms.

At any place of our development, you may use the `Print Assumptions` *ident* command in order to verify on which axiom the theorem *ident* may depend. The following example is extracted from Library hydras.Epsilon0.F_alpha, where we use the `coq-equations` plug-in (see Sect. 6.4 on page 122).

```
About F_zero_eqn.
```

```
F_zero_eqn : forall i : nat, F_ Zero i = S i
```

```
Print Assumptions F_zero_eqn.
```

```
Axioms:
FunctionalExtensionality.functional_extensionality_dep
  : forall (A : Type) (B : A -> Type) (f g : forall x : A, B x),
    (forall x : A, f x = g x) -> f = g
Eqdep.Eq_rect_eq.eq_rect_eq
  : forall (U : Type) (p : U) (Q : U -> Type) (x : Q p) (h : p = p),
    x = eq_rect p Q x p h
```

1.1.5 Main References

In our development, we adapt the definitions and prove many theorems which we found in the following articles.

- "Accessible independence results for Peano arithmetic" by Laurie Kirby and Jeff Paris [KP82]
- "Rapidly growing Ramsey Functions" by Jussi Ketonen and Robert Solovay [KS81]
- "The Termite and the Tower", by Will Sladek [Sla07]
- Chapter V of "Proof Theory" by Kurt Schütte [Sch77]

Warning: This document is *not* an introductory text for Coq, and there are many aspects of this proof assistant that are not covered. The reader should already have some basic experience with the Coq system. The Reference Manual and several tutorials are available on Coq page [Coq]. First chapters of textbooks like *Interactive Theorem Proving and Program Development* [BC04], *Software Foundations* [P$^+$] or *Certified Programming with Dependent Types* [Chl11] will give you the right background.

1.1.6 State of the Development

The Coq scripts herein are in constant development since our contribution [CC06] on notations for the ordinals ϵ_0 and Γ_0. We added new material : axiomatic definition of countable ordinals after Schütte [Sch77], combinatorial aspects of ϵ_0, after Ketonen and Solovay [KS81] and Kirby and Paris [KP82], recent Coq technology: type classes, equations, etc.

We are now working in order to make clumsy proofs more readable, simplify definitions, and "factorize" proofs as much as possible. Many possible improvements are suggested as "todo"s or "projects" in this text.

1.1.7 Contributions

Many thanks to Évelyne Contejean, Florian Hatat and Pascal Manoury for their contribution. Évelyne contributed libraries on the recursive path ordering (*rpo*) for proving the well-foundedness of our representation of ϵ_0 and Γ_0. Florian Hatat proved many useful lemmas on countable sets, which we used in our adaptation of Schütte's formalization of countable ordinals. Pascal Manoury is integrating the ordinal ω^ω into our hierarchy of ordinal notations.

Any form of contribution is welcome: correction of errors (typos and more serious mistakes), improvement of Coq scripts, proposition of inclusion of new chapters, and generally any comment or proposition that would help us. The text contains several *projects* which, when completed, may improve the present work. Please do not hesitate to bring your contribution!

1.1.8 Typographical Conventions

Quotations from our Coq source are displayed as follows:

```
Require Import Arith.

Definition square (n:nat) := n * n.

Lemma square_double : exists n:nat, n + n = square n.
Proof.
   exists 2.
```

Answers from Coq (including sub-goals, error messages, etc.) are displayed in slanted style with a different background color.

```
1 subgoal, subgoal 1 (ID 5)

 ============================
  2 + 2 = square 2
```

```
   reflexivity.
Qed.
```

1.1.9 Active Links

The links which appear in this pdf document lead are of three possible kinds of destination:

- Local links to the document itself,
- External links, mainly to Coq's page,
- Local links to pages generated by `coqdoc`. According to the current makefile (through the commands `make html` and `make pdf`), we assume that the page generated from a library `XXX/YYY.v` is stored as the relative address `../theories/html/hydras.XXX.YYY.html` (from the location of

the pdf) Thus, active links towards our Coq modules may be incorrect if you got this `pdf` document otherwise than by compiling the distribution available in `https://github.com/coq-community/hydra-battles`.

1.1.10 Alternative or Bad Definitions

Finally, we decided to include definitions or lemma statements, as well as tactics, that lead to dead-ends or too complex developments, with the following coloring. Bad definitions and encapsulation in modules called `Bad`, `Bad1`, etc.

```
Require Import Lia.

Module Bad.

Definition double (n:nat)  := n + 2.

Lemma lt_double : forall n:nat, n < double  n.
Proof.
   unfold double; lia.
Qed.

End Bad.
```

Likewise, alternative, but still unexplored definitions will be presented in modules `Alt`, `Alt1`, etc. Using these definitions is left as an implicit exercise.

```
Require Import Arith Lia Iterates.
Module Alt.
  Definition double (n : nat) := iterate S n n.
End Alt.
```

```
Lemma alt_double_ok n : Nat.double n = Alt.double n.
Proof.
  unfold Alt.double, Nat.double; induction n; cbn.
  -  trivial.
  -  rewrite <- iterate_rw, iterate_S_eqn, <- IHn; lia.
Qed.
```

1.2 Acknowledgements

Many thanks to David Ilcinkas, Sylvain Salvati, Alan Schmitt and Théo Zimmerman for their help on the elaboration of this document, and to the members of the *Formal Methods* team at laBRI for their helpful comments on an oral presentation of this work.

Many thanks also to the Coq development team, Yves Bertot, and members of the *Coq Club* for interesting discussions about the Coq system and the Calculus of Inductive Constructions.

The author of the present document wishes to express his gratitude to the late Patrick Dehornoy, whose talk was determinant for our desire to work on this topic.

I owe my interest in discrete mathematics and their relation to formal proofs and functional programming to Srecko Brlek. Equally, there is W. H. Burge's book "*Recursive Programming Techniques*" [Bur75] which was a great source of inspiration.

Chapter 2

Hydras and Hydra Games

This chapter is dedicated to the representation of hydras and rules of the hydra game in Coq's specification language: Gallina.

Technically, a *hydra* is just a finite ordered tree, each node of which has any number of sons. Note that, contrary to the computer science tradition, we will show the hydras with the heads up and the foot (i.e., the root of the tree) down. Fig. 2.1 represents such a hydra, which will be referred to as `Hy` in our examples (please look at the module Hydra.Hydra_Examples).

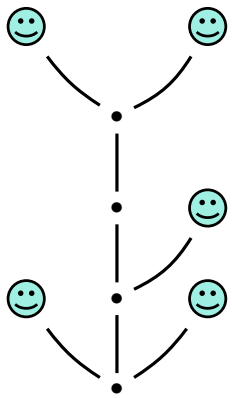

Figure 2.1: The hydra Hy

We use a specific vocabulary for talking about hydras. Table 2.2 shows the correspondance between our terminology and the usual vocabulary for trees in computer science.

The hydra `Hy` has a *foot* (below), five *heads*, and eight *segments*. We leave it to the reader to define various parameters such as the height, the size, the highest arity (number of sons of a node) of a hydra. In our example, these parameters have the respective values : 4, 9 and 3.

2.0.1 The Rules of the Game

A *hydra battle* is a fight between Hercules and the Hydra. More formally, a battle is a sequence of *rounds*. At each round:

Hydras	Finite rooted trees
foot	root
head	leaf
node	node
segment	(directed) edge
sub-hydra	subtree
daughter	immediate subtree

Figure 2.2: Translation from hydras to trees

- If the hydra is composed of just one head, the battle is finished and Hercules is the winner.
- Otherwise, Hercules chops off *one* head of the hydra,
  - If the head is at distance 1 from the foot, the head is just lost by the hydra, with no more reaction.
  - Otherwise, let us denote by r the node that was at distance 2 from the removed head in the direction of the foot, and consider the sub-hydra h' of h, whose root is r [1]. Let n be some natural number. Then h' is replaced by $n+1$ of copies of h' which share the same root r. The *replication factor* n may be different (and generally is) at each round of the fight. It may be chosen by the hydra, according to its strategy, or imposed by some particular rule. In many presentations of hydra battles, this number is increased by 1 at each round. In the following presentation, we will also consider battles where the hydra is free to choose its replication factor at each round of the battle[2].

Note that the description given in [KP82] of the replication process in hydra battles is also semi-formal.

> "From the node that used to be attached to the head which was just chopped off, traverse one segment towards the root until the next node is reached. From this node sprout n replicas of that part of the hydra (after decapitation) which is "above" the segment just traversed, i.e., those nodes and segments from which, in order to reach the root, this segment would have to be traversed. If the head just chopped off had the root of its nodes, no new head is grown. "

Moreover, we note that this description is in *imperative* terms. In order to build a formal study of the properties of hydra battles, we prefer to use a mathematical vocabulary, i.e., graphs, relations, functions, etc. Thus, the replication process will be represented as a binary relation on a data type `Hydra`, linking the state of the hydra *before* and *after* the transformation. A battle will thus be represented as a sequence of terms of type `Hydra`, respecting the rules of the game.

[1] h' will be called "the wounded part of the hydra" in the subsequent text. In Figures 2.4 on the next page and 2.6 on page 18, this sub-hydra is displayed in red.

[2] Let us recall that, if the chopped-off head was at distance 1 from the foot, the replication factor is meaningless.

2.0.2 Example

Let us start a battle between Hercules and the hydra `Hy` of Fig. 2.1.

At the first round, Hercules choses to chop off the rightmost head of `Hy`. Since this head is near the floor, the hydra loses this head. Let us call `Hy'` the resulting state of the hydra, represented in Fig. 2.3.

Next, assume Hercules choses to chop off one of the two highest heads of `Hy'`, for instance the rightmost one. Fig. 2.4 represents the rotten neck in dashed lines, and the part that will be replicated in red. Assume also that the hydra decides to add 4 copies of the red part[3]. We obtain a new state `Hy''` depicted in Fig. 2.5.

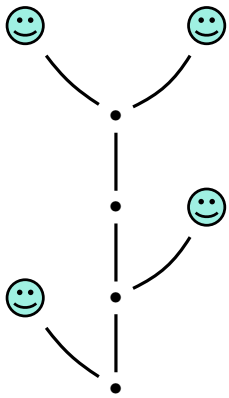

Figure 2.3: Hy': the state of Hy after one round

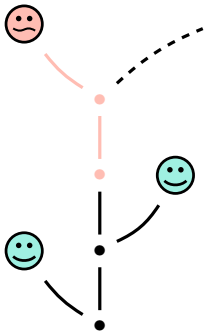

Figure 2.4: A second beheading

Figs. 2.6 and 2.7 on the next page represent a possible third round of the battle, with a replication factor equal to 2. Let us call `Hy'''` the state of the hydra after that third round.

[3]In other words, the replication factor at this round is equal to 4.

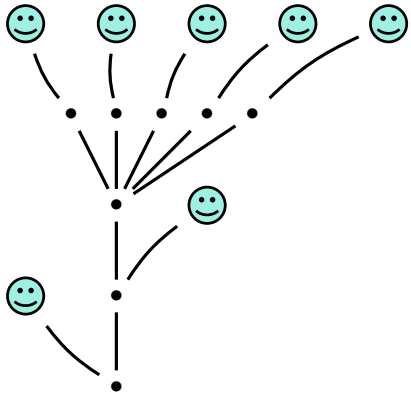

Figure 2.5: Hy", the state of Hy after two rounds

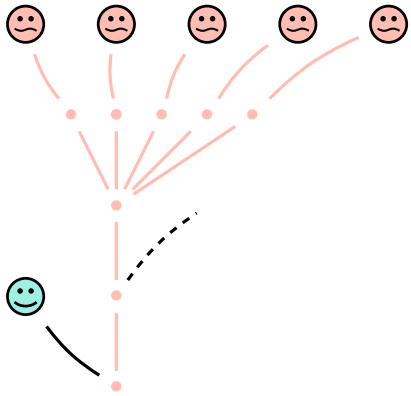

Figure 2.6: A third beheading (wounded part in red)

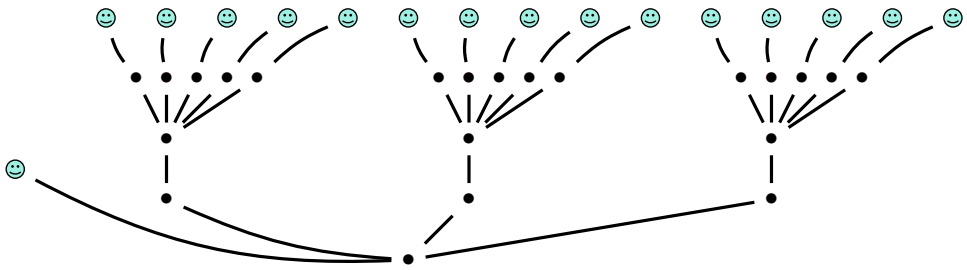

Figure 2.7: The configuration Hy"' of Hy

We leave it to the reader to guess the following rounds of the battle ...

2.1 Hydras and their Representation in *Coq*

In order to describe trees where each node can have an arbitrary (but finite) number of sons, it usual to define a type where each node carries a *forest*, *i.e* a list of trees (see for instance Chapter 14, pages 400-406 of [BC04]).

For this purpose, we define two mutual *ad-hoc* inductive types, where `Hydra` is the main type, and `Hydrae` is a helper for describing finite sequences of hydra.

From Module Hydra.Hydra_Definitions

```
Inductive Hydra : Set :=
| node :  Hydrae -> Hydra
with Hydrae : Set :=
| hnil : Hydrae
| hcons : Hydra -> Hydrae -> Hydrae.
```

Project 2.1 Look for an existing library on trees with nodes of arbitrary arity, in order to replace this ad-hoc type with something more generic.

Project 2.2 Another very similar representation could use the `list` type family instead of the specific type `Hydrae`:

```
Module Alt.

Inductive Hydra: Set :=
  hnode (daughters : list Hydra).

End Alt.
```

Using this representation, re-define all the constructions of this chapter. You will probably have to use patterns described for instance in [BC04] or the archives of the Coq-club [Coq].

Project 2.3 The type `Hydra` above describes hydras as *plane trees*, i.e., as drawn on a sheet of paper or computer screen. Thus, hydras are *oriented*, and it is appropriate to consider a *leftmost* or *rightmost* head of the beast. It could be interesting to consider another representation, in which every non-leaf node has a *multi-set* – not an ordered list – of daughters.

2.1.0.1 Abbreviations

We provide several notations for hydra patterns which occur often in our developments.

From Module Hydra.Hydra_Definitions

```
(** heads *)
Notation head := (node hnil).

(** nodes  with 1, 2 or 3 daughters *)
Notation hyd1 h := (node (hcons h hnil)).
Notation hyd2 h h' := (node (hcons h (hcons h' hnil))).
Notation hyd3 h h' h'' :=
                   (node (hcons h (hcons h' (hcons h'' hnil)))).
```

For instance, the hydra Hy of Figure 2.1 on page 15 is defined in *Gallina* as follows:

From Module Hydra.Hydra_Examples

```
Example Hy := hyd3 head
                   (hyd2
                      (hyd1
                         (hyd2 head head))
                      head)
                   head.
```

Hydras quite frequently contain multiple copies of the same pattern. The following functions will help us to describe and reason about replications in hydra battles.

From Module Hydra.Hydra_Definitions

```
Fixpoint hcons_mult (h:Hydra)(n:nat)(s:Hydrae):Hydrae :=
  match n with
  | O => s
  | S p => hcons h (hcons_mult h p s)
  end.

(** hydra with n copies of the same daughter *)

Definition hyd_mult h n :=
  node (hcons_mult h n hnil).
```

For instance, the hydra Hy'' of Fig 2.5 on page 18 can be defined in Coq as follows:

From Module Hydra.Hydra_Examples

```
Example Hy'' :=
     hyd2 head
          (hyd2 (hyd_mult (hyd1 head) 5)
                head).
```

2.1.0.2 Recursive Functions on type `Hydra`

When defining a recursive function over the type `Hydra`, one has to consider the three constructors `node`, `hnil` and `hcons` of the mutually inductive types `Hydra` and `Hydrae`. Let us define for instance the function which computes the number of nodes of any hydra:

From Module Hydra.Hydra_Definitions

```
Fixpoint hsize (h:Hydra) : nat :=
  match h with node l => S (lhsize l)
  end
with lhsize l : nat :=
  match l with hnil => 0
             | hcons h hs => hsize h + lhsize hs
  end.

 Compute hsize Hy.
```

```
 = 9
     : nat
```

Likewise, the *height* (maximum distance between the foot and a head) is defined by mutual recursion:

```
Fixpoint height  (h:Hydra) : nat :=
  match h with node l => lheight l
  end
with lheight l : nat :=
  match l with
  | hnil => 0
  | hcons h hs => Max.max (S (height h)) (lheight hs)
  end.
```

```
Compute height Hy.
```

```
 = 4
     : nat
```

Exercise 2.1 Define a function `max_degree: Hydra → nat` which returns the highest degree of a node in any hydra. For instance, the evaluation of the term `(max_degree Hy)` should return 3.

2.1.1 Induction Principles for Hydras

In this section, we show how induction principles are used to prove properties on the type `Hydra`. Let us consider for instance the following statement:

> " The height of any hydra is strictly less than its size. "

2.1.1.1 A failed Attempt

One may try to use the default tactic of proof by induction, which corresponds to an application of the automatically generated induction principle for type `Hydra`:

```
Hydra_ind :
forall P : Hydra -> Prop,
(forall h : Hydrae, P (node h)) -> forall h : Hydra, P h
```

Ler us start a simple proof by induction.

From Module Hydra.Hydra_Examples

```
Module Bad.

Lemma height_lt_size (h:Hydra) :
  height h <= hsize h.
Proof.
  induction h as [s].
```

```
1 subgoal, subgoal 1 (ID 11)

  s : Hydrae
  ============================
   height (node s) <= hsize (node s)
```

We might be tempted to do an induction on the sequence `s`:

```
 1 focused subgoal
(unfocused: 0), subgoal 1 (ID 19)

  h : Hydra
  s' : Hydrae
  IHs' : height (node s') <= hsize (node s')
  ============================
   height (node (hcons h s')) <= hsize (node (hcons h s'))
```

Note that the displayed subgoal does not contain any assumption on `h`, thus there is no way to infer any property about the height and size of the hydra `(hcons h t)`.

```
Abort.

End Bad.
```

2.1.1.2 A Principle of Mutual Induction

In order to get an appropriate induction scheme for the types `Hydra` and `Hydrae`, we can use Coq's command `Scheme`.

```
Scheme Hydra_rect2 := Induction for Hydra Sort Type
with Hydrae_rect2 := Induction for Hydrae Sort Type.
```

```
Check Hydra_rect2.
```

```
Hydra_rect2
 : forall (P : Hydra -> Type) (P0 : Hydrae -> Type),
   (forall h : Hydrae, P0 h -> P (node h)) ->
    P0 hnil ->
    (forall h : Hydra, P h ->
              forall h0 : Hydrae, P0 h0 -> P0 (hcons h h0)) ->
    forall h : Hydra, P h
```

2.1.1.3 A Correct Proof

Let us now use `Hydra_rect2` for proving that the height of any hydra is strictly less than its size. Using this scheme requires an auxiliary predicate, called `P0` in `Hydra_rect2`'s statement. Let us begin by defining an ad-hoc version of `List.Forall`.

From Module Hydra.Hydra_Examples

```
(** All elements of s satisfy P *)

Fixpoint h_forall (P: Hydra -> Prop) (s: Hydrae) :=
  match s with
      hnil => True
    | hcons h s' => P h /\ h_forall P s'
  end.
```

```
Lemma height_lt_size (h:Hydra) :
 height h < hsize h.
Proof.
  induction h using Hydra_rect2  with
  (P0 :=  h_forall (fun h =>  height h < hsize h)).
```

1. The first subgoal is as follows:

   ```
   h: Hydrae
   IHh : h_forall (fun h : Hydra => height h < hsize h) h
   ============================
    height (node s) < hsize (node s)
   ```

   This goal is easily solvable, using some arithmetic. We let the reader look at the source.

2. The second subgoal is trivial:

```
============================
  h_forall (fun h : Hydra => height h < hsize h) hnil
```

```
reflexivity.
```

3. Finally, the last subgoal is also easy to solve:

```
h : Hydra
h0 : Hydrae
IHh : height h < hsize h
IHh0 : h_forall (fun h : Hydra => height h < hsize h) h0
============================
 h_forall (fun h1 : Hydra => height h1 < hsize h1)
                (hcons h h0)
```

```
 split;auto.
Qed.
```

Exercise 2.2 It happens very often that, in the proof of a proposition of the form (∀ `h:Hydra,` P `h`), the predicate `P0` is (`h_forall` P). Design a tactic for induction on hydras that frees the user from binding explicitly `P0`, and solves trivial subgoals. Apply it for writing a shorter proof of `height_lt_size`.

2.2 Relational Description of Hydra Battles

In this section, we represent the rules of hydra battles as a binary relation associated with a *round*, i.e., an interaction composed of the two following actions:

1. Hercules chops off one head of the hydra
2. Then, the hydra replicates the wounded part (if the head is at distance ≥ 2 from the foot).

The relation associated with each round of the battle is parameterized by the *expected* replication factor (irrelevant if the chopped head is at distance 1 from the foot, but present for consistency's sake).

In our description, we will apply the following naming convention: if h represents the configuration of the hydra before a round, then the configuration of h after this round will be called h'. Thus, we are going to define a proposition (`round_n` n h h') whose intended meaning will be " the hydra h is transformed into h' in a single round of a battle, with the expected replication factor n ".

Since the replication of parts of the hydra depends on the distance of the chopped head from the foot, we decompose our description into two main cases, under the form of a bunch of [mutually] inductive predicates over the types `Hydra` and `Hydrae`.

The mutually exclusive cases we consider are the following:

- **R1**: The chopped off head was at distance 1 from the foot.
- **R2**: The chopped off head was at a distance greater than or equal to 2 from the foot.

2.2.1 Chopping off a Head at Distance 1 from the Foot (Relation R1)

If Hercules chops off a head near the floor, there is no replication at all. We use an auxiliary predicate `S0`, associated with the removing of one head from a sequence of hydras.

From ModuleHydra.Hydra_Definitions

```
Inductive S0 :  relation Hydrae :=
| S0_first : forall s, S0  (hcons head s) s
| S0_rest : forall  h s s',  S0  s s' ->
                            S0  (hcons h s) (hcons h s').

Inductive R1  :  Hydra -> Hydra -> Prop :=
| R1_intro : forall s s', S0 s s' -> R1 (node s) (node s').
```

2.2.1.1 Example

Let us represent in Coq the transformation of the hydra of Fig. 2.1 on page 15 into the configuration represented in Fig. 2.3.

From Module Hydra.Hydra_Examples

```
Example Hy_1 : R1 Hy Hy'.
Proof.
  split; right; right; left.
Qed.
```

2.2.2 Chopping off a Head at Distance ≥ 2 from the Foot (relation R2)

Let us now consider beheadings where the chopped-off head is at distance greater than or equal to 2 from the foot. All the following relations are parameterized by the replication factor n.

Let s be a sequence of hydras. The proposition (`S1 n s s'`) holds if s' is obtained by replacing some element h of s by $n+1$ copies of h', where the proposition (`R1 h h'`) holds, in other words, h' is just h, without the chopped-off head. `S1` is an inductive relation with two constructors that allow us to choose the position in s' of the wounded sub-hydra h.

From Module Hydra.Hydra_Definitions

```
Inductive S1 (n:nat)  : Hydrae -> Hydrae -> Prop :=
| S1_first : forall s h h' ,   R1 h h' ->
                  S1 n (hcons h s) (hcons_mult h' (S n) s)
| S1_next : forall h s s',  S1 n s s' ->
                   S1 n (hcons h s) (hcons h s').
```

The rest of the definition is structured as two mutually inductive relations on hydras and sequences of hydras. The first constructor of R2 describes the case where the chopped head is exactly at height 2. The others constructors allow us to consider beheadings at height strictly greater than 2.

From Module Hydra.Hydra_Definitions

```
Inductive R2 (n:nat)  :  Hydra -> Hydra -> Prop :=
| R2_intro : forall s s', S1 n s s' -> R2 n (node s) (node s')
| R2_intro_2 : forall s s', S2 n s s' -> R2 n (node s) (node s')

with S2 (n:nat) :  Hydrae -> Hydrae -> Prop :=
|  S2_first : forall h h' s ,
               R2 n  h h'  ->
               S2  n (hcons h s) (hcons h'  s)
|  S2_next  : forall h   r r',
               S2 n   r r' ->
               S2 n (hcons h r) (hcons h r').
```

2.2.2.1 Example

Let us prove the transformation of Hy' into Hy'' (see Fig. 2.5 on page 18). We use an experimental set of tactics for specifying the place where the interaction between Hercules and the hydra holds.

From Module Hydra.Hydra_Examples.

```
Example R2_example:  R2 4 Hy' Hy''.
Proof.
    (** move to 2nd sub-hydra (0-based indices) *) R2_up 1.
    (** move to first sub-hydra *)  R2_up 0.
    (** we're at distance 2 from the to-be-chopped-off head
        let's go to the first daughter,
        then chop-off the leftmost head *)  r2_d2  0 0.
Qed.
```

The reader is encouraged to look at all the successive subgoals of this example. *Please consider also exercise 2.5 on the facing page.*

2.2.3 Relation Associated with a Round

We combine the two cases above into a single relation. First, we define the relation (round_n n h h') where n is the expected number of replications (irrelevant in the case of an R1-transformation).

From Module Hydra.Hydra_Definitions

```
Definition round_n n h h' := R1 h h' \/ R2 n h h'.
```

By abstraction over n, we define a *round* (small step) of a battle:

```
Definition round h h' := exists n,  round_n n h h'.

Infix "-1->" := round (at level 60).
```

Project 2.4 Give a direct translation of Kirby and Paris's description of hydra battles (quoted on page 16) and prove that our relational description is consistent with theirs.

2.2.4 Rounds and Battles

Using library Relations.Relation_Operators, we define `round_plus`, the transitive closure of `round`, and `round_star`, the reflexive and transitive closure of `round`.

```
Definition round_plus := clos_trans_1n Hydra round.
Infix "-+->" := rounds (at level 60).

Definition round_star h h' := h = h' \/ round_plus h h'.
Infix "-*->" := round_star (at level 60).
```

Exercise 2.3 Prove the following lemma:

```
Lemma rounds_height : forall h h',
   h -+-> h' -> height h' <= height h.
```

Remark 2.1 Coq's library Coq.Relations.Relation_Operators contains three logically equivalent definitions of the transitive closure of a binary relation. This equivalence is proved in Coq.Relations.Operators_Properties .

Why three definitions for a single mathematical concept? Each definition generates an associated induction principle. According to the form of statement one would like to prove, there is a "best choice":

- For proving $\forall y, x\, R^+\, y \to P\, y$, prefer `clos_trans_n1`
- For proving $\forall x,\ x\, R^+\, y \to P\, x$, prefer `clos_trans_1n`
- For proving $\forall x\, y,\ x\, R^+\, y \to P\, x\, y$, prefer `clos_trans`,

But there is no "wrong choice" at all: the equivalence lemmas in Coq.Relations.Operators_Properties allow the user to convert any one of the three closures into another one before applying the corresponding elimination tactic. The same remark also holds for reflexive and transitive closures.

Exercise 2.4 Define a restriction of **round**, where Hercules always chops off the leftmost among the lowest heads.

Prove that, if h is not a simple head, then there exists a unique h' such that `h` is transformed into `h'` in one round, according to this restriction.

Exercise 2.5 (Interactive battles) Given a hydra `h`, the specification of a hydra battle for `h` is the type `{h':Hydra | h -*-> h'}`. In order to avoid long sequences of `split`, `left`, and `right`, design a set of dedicated tactics for the interactive building of a battle. Your tactics will have the following functionalities:

- Chose to stop a battle, or continue
- Chose an expected number of replications
- Navigate in a hydra, looking for a head to chop off.

Use your tactics for simulating a small part of a hydra battle, for instance the rounds which lead from `Hy` to `Hy'''` (Fig. 2.7 on page 18).

Hints:

- Please keep in mind that the last configuration of your interactively built battle is known only at the end of the battle. Thus, you will have to create and solve subgoals with existential variables. For that purpose, the tactic `eexists`, applied to the goal `{h':Hydra | h -*-> h'}` generates the subgoal `h -*-> ?h'`.
- You may use Gérard Huet's *zipper* data structure [Hue97] for writing tactics associated with Hercule's interactive search of a head to chop off.

2.2.5 Classes of Battles

In some presentations of hydra battles, e.g. [KP82, Bau08], the transformation associated with the i-th round may depend on i. For instance, in these articles, the replication factor at the i-th round is equal to i. In other examples, one can allow the hydra to apply any replication factor at any time. In order to be the most general as possible, we define the type of predicates which relate the state of the hydra before and after the i-th round of a battle.

From Module Hydra.Hydra_Definitions

```
Definition dep_round_t := nat -> Hydra -> Hydra -> Prop.

Class Battle :=  {battle_r : dep_round_t;
                  battle_inclusion : forall i h h',
                      battle_r i h h' -> round h h'}.
```

The most general class of battles is `free`, which allows the hydra to chose any replication factor at every step:

From Module Hydra.Hydra_Definitions

```
Program Instance free : Battle :=
  (Build_Battle ( fun _  h h' => round h h') _).
```

We chosed to call *standard* the kind of battles which appear most often in the litterature and correspond to an arithmetic progression of the replication factor : $0, 1, 2, 3, \ldots$

From Module Hydra.Hydra_Definitions

```
Program Instance standard : Battle := (Build_Battle round_n _).
Next Obligation.
  now exists i.
Defined.
```

2.2.6 Big Steps

Let B be any instance of class `Battle`. It is easy to define inductively the relation between the i-th and the j-th steps of a battle of type B.

From Module Hydra.Hydra_Definitions

```
Inductive battle (B:Battle) : nat -> Hydra -> nat -> Hydra -> Prop :=

| battle_1 : forall i h  h', battle_r   B i  h h' ->
                             battle B i h (S i) h'
| battle_n : forall i h  j h' h'',  battle_r  B i h h''  ->
                                    battle B (S i) h'' j h'  ->
                                    battle B i h j h'.
```

2.3 A Long Battle

In this section we consider a simple example of battle, starting with a small hydra, shown on figure 2.8, with a simple strategy for both players:

- Hercules chops off always the rightmost head of the hydra.
- The battle is standard: at the round number i, the expected replication is i.

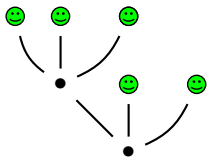

Figure 2.8: The hydra hinit

```
Definition hinit := hyd3 (hyd_mult head 3)  head head.
```

The lemma we would like to prove is "The considered battle lasts exactly N rounds", with N being a natural number we gave to guess.

But the battle is so long that no *test* can give us an estimation of its length, and we do need the expressive power of logic to compute this length. However, in order to guess this length, we made some experiments, computing with Gallina, Coq's functional programming language. Thus, we can consider this development as a collaboration of proof with computation.

In the following lines, we try to show faithfully how we found the value of the number N.

The complete proof is in file `../theories/html/hydras.Hydra.BigBattle.html`.

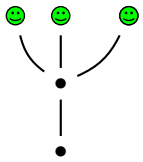

Figure 2.9: The hydra (hyd1 h3)

2.3.1 The beginning of Hostilities

During the two first rounds, our hydra loses its two rightmost heads. Thus just before the third round, it looks like in figure 2.9.

The following lemma is a formal description of these first rounds, in terms of the `battle` predicate.

```
Lemma L_0_2 : battle standard 0 hinit 2 (hyd1 h3).
```

2.3.2 Looking for Regularities

A first study with pencil and paper suggested us that, after three rounds, the hydra always looks like in figure 2.10 (with a variable number of subtrees of height 1 or 0). Thus, we introduce handy notations.

```
Notation h3 := (hyd_mult head 3).
Notation h2 := (hyd_mult head 2).
Notation h1 := (hyd1 head).

Definition hyd a b c :=
  node (hcons_mult h2  a
             (hcons_mult h1  b
                         (hcons_mult head c hnil))).
```

For instance Fig 2.10 shows the hydra (`hyd 3 4 2`). The hydra (`hyd 0 0 0`) is the "final" hydra of any terminating battle, i.e., a tree whith exactly one node and no edge.

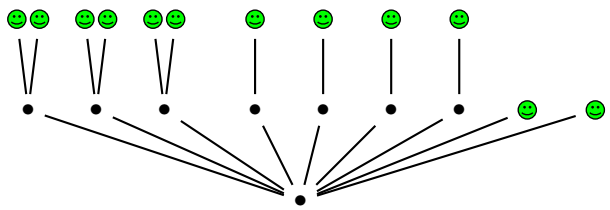

Figure 2.10: The hydra (hyd 3 4 2)

With these notations, we get a formal description of the first three rounds.

```
Lemma L_2_3 : battle standard 2 (hyd1 h3)  3 (hyd 3 0 0).

Lemma L_0_3 : battle standard 0 hinit 3 (hyd 3 0 0).
```

2.3.3 Computing ...

In order to study *experimentally* the different configurations of the battle, we will use a simple datatype for representing the states as tuples composed of the round number, and the respective number of daughters `h2`, `h1`, and heads of the current hydra.

```
Record state : Type :=
   mks {round: nat ; n2 : nat ; n1 : nat ; nh : nat}.
```

The following function returns the next configurarion of the game. Note that this function is defined only for making experiments and is not "certified". Formal proofs about our battle will only start with the lemma `lemma:step-battle`, page 33.

```
Definition next (s : state) :=
  match s with
  | mks round a b (S c) => mks (S round) a b c
  | mks round a (S b) 0 => mks (S round) a b (S round)
  | mks round (S a) 0 0 => mks (S round) a (S round) 0
  | _ => s
  end.
```

We can make bigger steps through iterations of `next`. The functional `iterate`, similar to Standard Library's `Nat.iter`, is defined and studied in Prelude.Iterates.

```
Fixpoint iterate {A:Type}(f : A -> A) (n: nat)(x:A) :=
  match n with
  | 0 => x
  | S p => f (iterate  f p x)
  end.
```

The following function computes the state of the battle at the n-th round.

```
Definition test n := iterate next (n-3) (mks 3 3 0 0).

Compute test 3.
   (**
     = {| round := 3; n2 := 3; n1 := 0; nh := 0 |}
     : state
    *)

Compute test 4.
 (*
  = {| round := 4; n2 := 2; n1 := 4; nh := 0 |}
     : state
 *)

Compute test 5.
 (*
   = {| round := 5; n2 := 2; n1 := 3; nh := 5 |}
```

```
     : state
 *)

Compute test 2000.
(*
  = {| round := 2000; n2 := 1; n1 := 90; nh := 1102 |}
     : state
*)
```

The battle we are studying seems to be awfully long. Let us concentrate our tests on some particular events : the states where `nh` $= 0$. From the value of `test 5`, it is obvious that at the 10-th round, the counter `nh` is equal to zero.

```
 Compute test 10.
(*

    = {| round := 10; n2 := 2; n1 := 3; nh := 0 |}
     : state
*)
```

Thus, $(1 + 11)$ rounds later, the `n1` field is equal to 2, and `nh` to 0.

```
Compute test 22.
(*

 = {| round := 22; n2 := 2; n1 := 2; nh := 0 |}
     : state
*)
```

```
 Compute test 46.
(*

 = {| round := 46; n2 := 2; n1 := 1; nh := 0 |}
     : state

*)

Compute test 94.

(*

 = {| round := 94; n2 := 2; n1 := 0; nh := 0 |}
     : state

*)
```

Next round, we decrement `n2` and set `n1` to 95.

```
 Compute test 95.

(*
```

```
  = {| round := 95; n2 := 1; n1 := 95; nh := 0 |}
     : state

*)
```

We now have some intuition of the sequence. It looks like the next "nh=0" event will happen at the $192 = 2(95 + 1)$-th round, then at the $2(192 + 1)$-th round, etc.

```
Definition doubleS (n : nat) := 2 * (S n).

Compute test (doubleS 95).

(**
 = {| round := 192; n2 := 1; n1 := 94; nh := 0 |}
     : state
 *)


Compute test (iterate doubleS 2 95).

(*
  = {| round := 386; n2 := 1; n1 := 93; nh := 0 |}
     : state
*)
```

2.3.4 Proving ...

We are now able to reason about the sequence of transitions defined by our hydra battle. Instead of using the data-type `state` we study the relationship between different configurations of the battle.

Let us define a binary relation associated with every round of the battle. In the following definition `i` is associated with the round number (or date, if we consider a discrete time), and `a`, `b`, `c` respectively associated with the number of `h2`, `h1` and heads connected to the hydra's foot.

```
Inductive one_step (i: nat) :
  nat -> nat -> nat -> nat -> nat -> nat -> Prop :=
| step1: forall a b c, one_step i a b (S c) a b c
| step2:  forall a b, one_step i a (S b) 0 a b (S i)
| step3: forall a, one_step i (S a) 0 0 a (S i) 0.
```

The relation between `one_step` and the rules of hydra battle is asserted by the following lemma.

```
Lemma step_battle : forall i a b c a' b' c',
   one_step i a b c a' b' c' ->
   battle standard i (hyd  a b c)  (S i) (hyd a' b' c').
```

Next, we define "big steps" as the transitive closure of `one_step`, and reachability (from the initial configuration of figure 2.8 at time 0).

```
 Inductive steps : nat -> nat -> nat -> nat ->
                   nat -> nat -> nat -> nat -> Prop :=
| steps1 : forall i a b c a' b' c',
    one_step i a b c a' b' c' -> steps i a b c (S i) a' b' c'
| steps_S : forall i a b c j a' b' c' k a'' b'' c'',
    steps i a b c j a' b' c' ->
    steps j a' b' c' k a'' b'' c'' ->
    steps i a b c k  a'' b'' c''.

Definition reachable (i a b c : nat) : Prop :=
  steps 3 3 0 0 i a b c.
```

The following lemma establishes a relation between `steps` and the predicate `battle`.

```
 Lemma steps_battle : forall i a b c j a' b' c',
   steps i a b c j a' b' c' ->
   battle standard i (hyd  a b c)   j  (hyd a' b' c').
```

Thus, any result about `steps` will be applicable to standard battles. Using the predicate `steps` our study of the length of the considered battle can be decomposed into three parts:

1. Characterization of regularities of some events
2. Study of the beginning of the battle
3. Computing the exact length of the battle.

First, we prove that, if at round i the hydra is equal to (`hyd a (S b) 0`), then it will be equal to (`hyd a b 0`) at the $2(i+1)$-th round.

```
Lemma LS : forall c a b i,  steps i a b (S c) (i + S c) a b 0.
Proof.
  induction c.
 -   intros;  replace (i + 1) with (S i).
     + repeat constructor.
     + ring.
 -  intros; eapply  steps_S.
   +   eleft;   apply rule1.
   +   replace (i + S (S c)) with (S i + S c) by ring;  apply IHc.
Qed.
```

```
Lemma doubleS_law : forall  a b i, steps i a (S b) 0 (doubleS i) a b 0.
Proof.
  intros;  eapply steps_S.
  +   eleft;   apply step2.
  +   unfold doubleS; replace (2 * S i) with (S i + S i) by ring;
        apply LS.
Qed.
```

```
Lemma reachable_S  : forall i a b, reachable i a (S b) 0 ->
                                   reachable (doubleS i) a b 0.
Proof.
  intros; right with  (1 := H); apply doubleS_law.
Qed.
```

From now on, the lemma `reachable_S` allows us to watch larger steps of the battle.

```
 Lemma L4 : reachable 4 2 4 0.
Proof.
  left; constructor.
Qed.
```

```
Lemma L10 : reachable 10 2 3 0.
Proof.
  change 10 with (doubleS 4).
  apply reachable_S, L4.
Qed.
```

```
Lemma L22 : reachable 22 2 2 0.
Proof.
  change 22 with (doubleS 10).
  apply reachable_S, L10.
Qed.
```

```
Lemma L46 : reachable 46 2 1 0.
Proof.
  change 46 with (doubleS 22); apply  reachable_S, L22.
Qed.
```

```
Lemma L94 : reachable 94 2 0 0.
Proof.
  change 94 with (doubleS 46); apply reachable_S, L46.
Qed.
```

```
Lemma L95 : reachable 95 1 95 0.
Proof.
  eapply steps_S.
  -  eexact L94.
  -  repeat constructor.
Qed.
```

2.3.5 Giant Steps

We are now able to make bigger steps in the simulation of the battle. First, we iterate the lemma `reachable_S`.

```
Lemma Bigstep : forall b i a , reachable i a b 0 ->
                               reachable (iterate doubleS b i) a 0 0.
```

```
Proof.
 induction b.
 -  trivial.
 -  intros;  simpl;   apply reachable_S in H.
    rewrite <- iterate_comm; now apply IHb.
Qed.
```

Applying lemmas `BigStep` and `L95` we make a first jump.

```
Definition M := (iterate doubleS 95 95).

Lemma L2_95 : reachable M 1 0 0.
Proof.
  apply Bigstep,  L95.
Qed.
```

Figure 2.11 represents the hydra at the M-th round. At the $(M+1)$-th round, it will look like in fig 2.12.

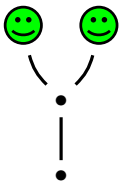

Figure 2.11
The state of the hydra after M rounds.

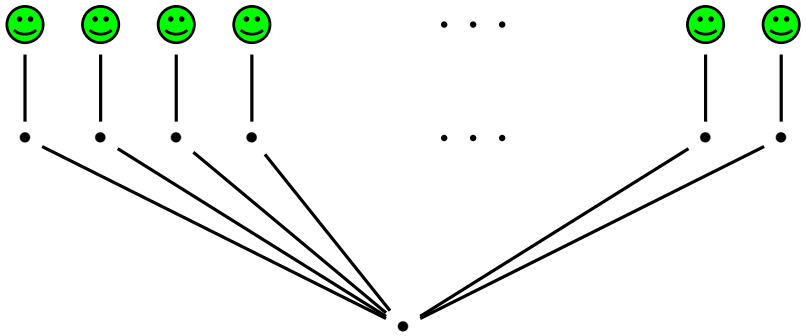

Figure 2.12
The state of the hydra after $M+1$ rounds (with $M+1$ heads).

```
Lemma L2_95_S : reachable (S M) 0 (S M) 0.
Proof.
  eright.
  - apply L2_95.
  -  left; constructor 3.
Qed.
```

Then, applying once more the lemma `BigStep`, we get the exact time when Hercules wins!

```
Definition N :=   iterate doubleS (S M) (S M).

Theorem   SuperbigStep : reachable N  0 0 0 .
Proof.
  apply Bigstep, L2_95_S.
Qed.
```

We are now able to prove formally that the considered battle is composed of N steps.

```
Lemma Almost_done :
  battle standard 3 (hyd 3 0 0) N (hyd 0 0 0).
Proof.
  apply steps_battle, SuperbigStep.
Qed.

Theorem Done :
  battle standard 0 hinit N head.
Proof.
  eapply battle_trans.
  -   apply Almost_done.
  -  apply L_0_3.
Qed.
```

2.3.6 A Minoration Lemma

Now, we would like to get an intuition of how big the number N is. For that purpose, we use a minoration of the function `doubleS` by the function (`fun n => 2 * n`).

```
Definition exp2 n := iterate (fun n => 2 * n) n 1.
```

Using some facts (proven in hydras.Hydra.BigBattle),we get several minorations.

```
Lemma minoration_0 : forall n,  2 * n <= doubleS n.

Lemma minoration_1 : forall n x, exp2 n * x <= iterate doubleS n x.

Lemma minoration_2 : exp2 95 * 95 <= M.

Lemma minoration_3 : exp2 (S M) * S M <= N.

Lemma minoration : exp2 (exp2 95 * 95) <= N.
```

The number N is greater than or equal to $2^{2^{95}\times 95}$. If we wrote N in base 10, N would require at least 10^{30} digits!

2.4 Generic Properties

The example we just studied shows that the termination of any battle may take a very long time. If we want to study hydra battles in general, we have to consider any hydra and any strategy, both for Hercules and the hydra itself. So, we first give some definitions, generally borrowed from transition systems vocabulary (see [Tel00] for instance).

2.4.1 Liveliness

Let B be an instance of `Battle`. We say that B is *alive* if for any configuration (i, h), where h is not a head, there exists a further step in class B.

From Module Hydra.Hydra_Definitions

```
Definition Alive (B : Battle) :=
  forall i h,
     h <> head -> {h' : Hydra |  B i h h'}.
```

The theorems `Alive_free` and `Alive_standard` of the module `../theories/html/hydras.Hydra.Hydra_Theorems.html` show that the classes `free` and `standard` satisfy this property.

```
Theorem Alive_free: Alive free.

Theorem Alive_standard: Alive standard.
```

Both theorems are proved with the help of the following strongly specified function:

From Module Hydra.Hydra_Lemmas

```
Definition  next_round_dec n :
 forall h , (h = head) + {h' : Hydra & {R1 h h'} + {R2  n h  h'}}.
```

2.4.2 Termination

The termination of all battles is naturally expressed by the predicate `well_founded` defined in the module Coq.Init.Wf of the Standard Library.

```
Definition Termination :=  well_founded (transp _ round).
```

Let B be an instance of class `Battle`. A *variant* for B consists in a well-founded relation $<$ on some type `A`, and a function (also called a *measure*) `m:Hydra->A` such that for any successive steps (i, h) and $(1 + i, h')$ of a battle in B, the inequality $m(h') < m(h)$ holds.

From Module Hydra.Hydra_Definitions

```
Class Hvariant {A:Type}{Lt:relation A}(Wf: well_founded Lt)(B : Battle)
  (m: Hydra -> A):   Prop :=
  {variant_decr :forall i h h',
      h <> head ->
      battle_r  B i  h h' -> Lt (m h') (m h)}.
```

Exercise 2.6 Prove that, if there is an instance of (`Hvariant Lt wf_Lt` B m), then there exists no infinite battle in B.

2.4.3 A Small Proof of Impossibility

When one wants to prove a termination theorem with the help of a variant, one has to consider first a well-founded set $(A, <)$, then a strictly decreasing measure on this set. The following two lemmas show that if the order structure $(A, <)$ is too simple, it is useless to look for a convenient measure, which simply no exists. Such kind of result is useful, because it saves you time and effort.

The best known well-founded order is the natural order on the set $\mathbb{N}$ of natural numbers (the type `nat` of Standard library). It would be interesting to look for some measure m : `nat`→`nat` and prove it is a variant.

Unfortunately, we can prove that *no* instance of class (`WfVariant round Peano.lt` m) can be built, where m is *any* function of type `Hydra` → `nat`.

Let us present the main steps of that proof, the script of which is in the module Hydra/Omega_Small.v [4].

Let us assume there exists some variant m from `Hydra` into `nat` for proving the termination of all hydra battles.

```
Section Impossibility_Proof.
 Variable m : Hydra -> nat.
 Hypothesis Hvar : Hvariant lt_wf free m.
```

We define an injection from the type `nat` into `Hydra`. For any natural number i, $\iota(i)$ is the hydra composed of a foot and $i+1$ heads at height 1. For instance, Fig. 2.13 represents the hydra $\iota(3)$.

Figure 2.13: The hydra $\iota(3)$

```
  Let iota (i: nat) := hyd_mult head (S i).
```

Let us consider now some hydra `big_h` out of the range of the injection ι (see Fig. 2.14 on the following page).

```
  Let big_h := hyd1 (hyd1 head).
```

Using the functions m and ι, we define a second hydra `small_h`, and show there is a one-round battle that transforms `big_h` into `small_h`. Please note that, due to the hypothesis `Hvar`, we are interested in the termination of *free* battles. There is no problem to consider a round with (`m big_h`) as the replication factor.

[4] The name of this file means "the ordinal ω is too small for proving the termination of [free] hydra battles ". In effect, the elements of ω, considered as a set, are just the natural numbers (see next chapter for more details)

Figure 2.14
The hydra `big_h`.

```
Let small_h := iota (m big_h).

Fact big_to_small : big_h -1-> small_h.
Proof.
     exists (m big_h); right; repeat constructor.
Qed.
```

But, by hypothesis, m is a variant. Hence, we infer the following inequality.

```
Lemma m_lt : m small_h < m big_h.
```

In order to get a contradiction, it suffices to prove the inequality `m big_h <= m small_h`, i.e., `m big_h <= m (iota (m big_h))`.
More generally, we prove the following lemma:

```
Lemma m_ge : forall i:nat, i <= m (iota i).
```

Intuitively, it means that, from any hydra of the form (`iota` i), the battle will take (at least) i rounds. Thus the associated measure cannot be less than i. Technically, we prove this lemma by Peano induction on i.

- The base case $i = 0$ is trivial
- Otherwise, let i be any natural number and assume the inequality $i \leq m(\iota(i))$.
    1. But the hydra $\iota(S(i))$ can be transformed in one round into $\iota(i)$ (by losing its righmost head, for instance)
    2. Since m is a variant, we have $m(\iota(i)) < m(\iota(S(i)))$, hence $i < m(\iota(S(i)))$, which implies $S(i) \leq m(\iota(S(i)))$.

Then our proof is almost finished.

```
Theorem Contradiction : False.
Proof.
 apply (Nat.lt_irrefl (m big_h));
   apply  Lt.le_lt_trans with (m small_h).
  - apply m_ge.
  - apply m_lt.
```

```
Qed.

End Impossibility_Proof.
```

Exercise 2.7 Prove that there exists no variant m from `Hydra` into `nat` for proving the termination of all *standard* battles.

2.4.3.1 Conclusion

In order to build a variant for proving the termination of all hydra battles, we need to consider order structures more complex than the usual order on type `nat`. The notion of *ordinal number* provides a catalogue of well-founded order types. For a reasonably large bunch of ordinal numbers, *ordinal notations* are data-types which allow the Coq user to define functions, to compute and prove some properties, for instance by reflection.

The next chapter is dedicated to a generic implementation of ordinal notations, and chapter 4 to a proof of termination of all hydra battles with the help of an ordinal notation for the interval $[0, \epsilon_0)$.

Chapter 3

Introduction to Ordinal Numbers and Ordinal Notations

The proof of termination of all hydra battles presented in [KP82] is based on *ordinal numbers*. From a mathematical point of view, an ordinal is a representant of an equivalence class for isomorphims of totally ordered well-founded sets.

More intuitively, let us have a look at Figure 3.1. It presents a small sequence of ordinal numbers, which extends the sequence of natural numbers.

$$
\begin{array}{l}
0,\ 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, \ldots \\
\omega,\ \omega + 1, \omega + 2, \omega + 3, \ldots \\
\omega \times 2,\ \omega \times 2 + 1, \ldots, \omega \times 3,\ \omega \times 3 + 1, \ldots, \omega \times 4, \ldots, \\
\omega^2, \ldots, \omega^2 \times 42, \ldots, \omega^3, \ldots, \omega^4, \omega^4 + 1, \ldots, \\
\omega^\omega, \ldots, \omega^\omega + \omega^7 \times 8, \ldots, \omega^\omega \times 2, \omega^\omega \times 2 + 1, \ldots, \\
\omega^{\omega^\omega}, \ldots, \omega^{\omega^\omega} + \omega^\omega \times 42 + \omega^{55} + \omega, \ldots, \omega^{\omega^{\omega+1}}, \omega^{\omega^{\omega+1}} + 1, \ldots \\
\epsilon_0 (= \omega^{\epsilon_0}), \epsilon_0 + 1, \epsilon_0 + 2, \epsilon_0 + 3, \ldots, \\
\epsilon_1, \ldots, \epsilon_2, \ldots, \epsilon_\omega, \ldots \\
\Gamma_0, \Gamma_0 + 1, \Gamma_0 + 2, \Gamma_0 + 3, \ldots, \Gamma_0 + \omega, \ldots, \\
\ldots
\end{array}
$$

Figure 3.1: A short overview of the sequence of ordinal numbers

Let us comment some features of this figure:

- The ordinals are listed in a strictly increasing order.
- Dots : "..." stand for an infinite sequence of ordinals, not shown for lack of space. For instance, the ordinal 42 is not shown in the first line, but it exists, between 17 and ω.
- Each ordinal printed in black is the immediate successor of another ordinal. We call it a *successor* ordinal. For instance, 12 is the successor of 11, and $\omega^4 + 1$ the successor of ω^4.
- Ordinals (displayed in red) that follow immediately dots are called *limit ordinals*. With respect to the order induced by this sequence, any limit ordinal α is the least upper bound of the set $\mathbb{O}_\alpha$ of all ordinals strictly less than α.
- For instance ω is the least upper bound of the set of all finite ordinals (in the first line). It is also the first limit ordinal, and the first infinite ordinal, in the sense that the set $\mathbb{O}_\omega$ is infinite.
- The ordinal ϵ_0 is the first number which is equal to its own exponential of base ω. It plays an important role in proof theory, and is particularly studied in chapters 4 to 6.
- Any ordinal is either the ordinal 0, a successor ordinal, or a limit ordinal.

3.1 The Mathematical Point of View

3.1.1 Well-ordered Sets

Let us start with some definitions. A *well-ordered set* is a set provided with a binary relation $<$ which has the following properties.

irreflexivity : $\forall x \in A, x \not< x$

transitivity : $\forall x\, y\, z \in A, x < y \Rightarrow y < z \Rightarrow x < z$

trichotomy : $\forall x\, y \in A, x < y \vee x = y \vee y < x$

well foundedness : $<$ is well-founded (every element of A is accessible)[1].

The best known examples of well-ordered sets are the set $\mathbb{N}$ of natural numbers (with the usual order $<$), as well as any finite segment $[0, i) = \{j \in \mathbb{N} \mid j < i\}$. The disjoint union of two copies of $\mathbb{N}$, *i.e.* the set $\{0, 1\} \times \mathbb{N}$ is also well-ordered, with respect to the order below:

$$(i, j) < (i, k) \text{ if } j < k$$
$$(0, k) < (1, l) \text{ for any } k \text{ and } l$$

[1] In classical mathematics, we would say that there is no infinite sequence $a_1 > a_2 > \ldots a_n > a_{n+1} \ldots$ in A. Please refer to any documenation on Coq for having more details on well-foundedness and accessibility.

3.1.2 Ordinal Numbers

Let $(A, <_A)$ and $(B, <_B)$ two well-ordered sets. A and B are said to have *the same order type* if there exists a strictly monotonous bijection b from A to B, *i.e.* which verifies the proposition $\forall x\, y \in A, x <_A y \Rightarrow b(x) <_B b(y)$.

Having the same order type is an equivalence relation between well-ordered sets. Ordinal numbers (in short *ordinals*) are descriptions (*names*) of the equivalence classes. For instance, the order type of $(\mathbb{N}, <)$ is associated with the ordinal called ω, and the order we considered on the disjoint union of $\mathbb{N}$ and itself is named $\omega + \omega$.

In a set-theoretic framework, one can consider any ordinal α as a well-ordered set, whose elements are just the ordinals strictly less than α, *i.e.* the *segment* $\mathbb{O}_\alpha = [0, \alpha)$. So, one can speak about *finite*, *infinite*, *countable*, etc., ordinals. Nevertheless, since we work within type theory, we do not identify ordinals as sets of ordinals, but the correspondance between ordinals and sets of ordinals is the function that maps α to $\mathbb{O}_\alpha$. For instance $\mathbb{O}_\omega = \mathbb{N}$, and $\mathbb{O}_7 = \{0, 1, 2, 3, 4, 5, 6\}$.

We cannot cite all the litterature published on ordinals since Cantor's book [Can55], and leave it to the reader to explore the bibliography.

3.2 Ordinal Numbers in Coq

Two kinds of representation of ordinals are defined herein.

- A "mathematical" representation of the set of countable ordinal numbers, afer Kurt Schütte [Sch77]. This representation uses several (hopefully harmless) axioms. We use it as a reference for proving the correctness of ordinal notations.
- A family of *ordinal notations*, *i.e.* data types used to represent segments $[0, \mu)$, where μ is some countable ordinal. Each ordinal notation is defined inside the Calculus of Inductive Constructions (without axioms). Many functions are defined, allowing proofs by computation. Note that proofs of correctness of a given ordinal notation with respect to Schütte's model obviously use axioms. Please execute the `Print Assumptions` command in case of doubt.

3.3 Countable Ordinals

Chapter 7 of this document presents an adaptation to Coq of an axiomatization in classical logic of the set of countable ordinals by K. Schütte [Sch77]. That formalization is quite complex, technical and unshamedly non-constructive, so we put its description in the last chapter of this document.

Please note that Schütte considers the (uncountable) set $\mathbb{O}$ of all countable ordinals. This set is well ordered (which is one of Schütte's axioms), and associates to any ordinal α the segment $\mathbb{O}_\alpha$ of all ordinals strictly less than α.

In our adaptation to Coq, we declare a type `Ord`, a binary relation `lt` (with infix notation `"_<_"`, and assume Schütte's axiom. In Chapter 7, we derive some interesting properties of countable ordinals from these axioms. It is interesting to compare proofs of a given property (for instance the associativity of addition)

in the computatuinal framework of some ordinal notation, and in the axiomatic model of Schütte.

3.4 Ordinal Notations

Fortunately, the ordinals we need for studying hydra battles are much simpler than Schütte's, and can be represented as quite simple data types in Gallina. So, we will use *ordinal notations* (also called *[ordinal] notation systems*).

Let α be some (countable) ordinal; in Coq terms, we call ordinal notation for α a structure composed of:

- A data type A for representing all ordinals strictly below α,
- A well founded order $<$ on A,
- A correct function for comparing two ordinals. Note that the reflexive closure of $<$ is thus a *total order*.

Such a structure can be proved correct relatively to another ordinal notation or to Schütte's model.

Ordered Types

The library Coq.Classes.RelationClasses contains some definitions and facts about binary relations, among them strict orders.

```
Variable A: Type.

  Class StrictOrder (R : relation A) : Prop := {
    StrictOrder_Irreflexive :> Irreflexive R ;
    StrictOrder_Transitive :> Transitive R }.
```

3.4.1 A Class for Ordinal Notations

The following class definition, parameterized with a type A, a binary relation `lt` on A, specifies that `lt` is a well-founded strict order. The reflexive closure of `lt`, (called `le`, for "less or equal than") is a total decidable order, implemented through a comparison function `compare`. The correctness of this function is expressed through Stdlib's type `Datatypes.CompareSpec`.

```
Inductive CompareSpec (Peq Plt Pgt : Prop) : comparison -> Prop :=
    CompEq : Peq -> CompareSpec Peq Plt Pgt Eq
  | CompLt : Plt -> CompareSpec Peq Plt Pgt Lt
  | CompGt : Pgt -> CompareSpec Peq Plt Pgt Gt
```

From Library OrdinalNotations.Definitions

```
Class ON {A:Type}(lt: relation A)
      (compare : A -> A -> comparison)  :=
  {
  sto :> StrictOrder lt;
```

```
  wf : well_founded lt;
  compare_correct :
    forall alpha beta:A,
      CompareSpec (alpha=beta) (lt alpha beta) (lt beta alpha)
                  (compare alpha beta);
  }.
```

The following definitions allow us to make implicit several guessable arguments.

```
Definition on_t  {A:Type}{lt: relation A}
            {compare : A -> A -> comparison}
            {on : ON lt compare} := A.

Definition ON_compare {A:Type}{lt: relation A}
            {compare : A -> A -> comparison}
            {on : ON lt compare} := compare.

Definition ON_lt {A:Type}{lt: relation A}
           {compare : A -> A -> comparison}
           {on : ON lt compare} := lt.
Infix "o<" := ON_lt : ON_scope.

Definition ON_le  {A:Type}{lt: relation A}
           {compare : A -> A -> comparison}
           {on : ON lt compare} :=
  clos_refl _ ON_lt.

Infix "o<=" := ON_le : ON_scope.
```

Remark 3.1 The infix notations `o<` and `o<=` were defined in order to make apparent the distinction between the various notation scopes that may co-exist in a same statement. So the infix `<` and `<=` are reserved to the natural numbers. In the mathematical formulas, we still use $<$ and $\leq$ for comparing ordinals.

3.4.2 Ordinal Notations and Measures for Proving Termination

The following lemma (together with the type class mechnism) allows us to use simply measures towards an ordinal notation. It is just an application of the libraries `Coq.Wellfounded.Inverse_Image` and `Coq.Wellfounded.Inclusion`.

```
Definition measure_lt {A:Type}{lt: relation A}
            {compare : A -> A -> comparison}
            {on : ON lt compare}
            {B : Type} (m : B -> A) : relation B :=
             fun x y => on_lt (m x) (m y).

Lemma wf_measure  {A:Type}(lt: relation A)
            {compare : A -> A -> comparison}
            {on : ON lt compare}
```

```
            {B : Type}
            (m : B -> A):  well_founded (measure_lt m).
```

A simple example of application is given in Sect. 3.7.2 on page 54.

3.5 Examples of Ordinal Notations

3.5.1 The Ordinal ω

The simplest example of ordinal notation is built over the type `nat` of Coq's standard library. We have only to apply already proven lemmas about Peano numbers.

From Library OrdinalNotations.ON_Omega

```
Global Instance Omega : ON  Peano.lt Nat.compare.
Proof.
 split.
 - apply Nat.lt_strorder.
 - apply Wf_nat.lt_wf.
 - apply Nat.compare_spec.
Qed.
```

3.5.2 Sum of two Ordinal Notations

Let `NA` and `NB` be two ordinal notations, on the respective types `A` and `B`.

We consider a new strict order on the disjoint sum of the associated types, by putting all elements of `A` before the elements of `B` (thanks to Standard Library's relation operator `le_AsB`).

From Library Relations.Relation_Operators.

```
Inductive
le_AsB (A B : Type) (leA : A -> A -> Prop) (leB : B -> B -> Prop)
  : A + B -> A + B -> Prop :=
| le_aa : forall x y : A, leA x y -> le_AsB A B leA leB (inl x) (inl y)
| le_ab : forall (x : A) (y : B), le_AsB A B leA leB (inl x) (inr y)
| le_bb : forall x y : B, leB x y -> le_AsB A B leA leB (inr x) (inr y)
```

From Library OrdinalNotations.ON_plus

```
Section Defs.

  Context `(ltA: relation A)
          (compareA : A -> A -> comparison)
          (NA: ON ltA compareA).
  Context `(ltB: relation B)
          (compareB : B -> B -> comparison)
          (NB: ON ltB compareB).


Definition t := (A + B)%type.
```

```
Arguments inl  {A B} _.
Arguments inr  {A B} _.

Definition lt : relation t := le_AsB _ _ ltA ltB.
```

Before building an instance of `ON`, we have to define a comparison function.

```
Definition compare (alpha beta: t) : comparison :=
   match alpha, beta with
     inl _, inr _ => Lt
   | inl a, inl a' => compareA a a'
   | inr b, inr b' => compareB b b'
   | inr _, inl _ => Gt
  end.

Lemma compare_correct alpha beta :
    CompareSpec (alpha = beta) (lt alpha beta) (lt beta alpha)
                            (compare alpha beta).
```

The Lemma `Wellfounded.Disjoint_Union.wf_disjoint_sum` of Standard Library helps us to prove that our order `lt` is well-founded.

```
Global Instance ON_plus : ON lt compare.
Proof.
  split.
  - apply lt_strorder.
  -  apply lt_wf.
  - apply compare_correct.
Qed.
```

3.5.3 The Ordinal $\omega + \omega$

The ordinal $\omega+\omega$ (also known as $\omega \times 2$) may be represented as the concatenation of two copies of ω (Figure 3.2).

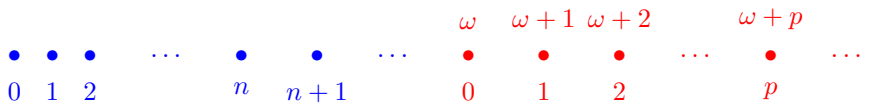

Figure 3.2: $\omega + \omega$

We can define this notation in Coq as an instance of `ON_plus`.

From Module OrdinalNotations.ON_Omega_plus_omega

```
Definition Omega_plus_Omega := ON_plus Omega Omega.

Existing Instance Omega_plus_Omega.
Definition t := @ON_plus.t nat nat.
```

```
Example ex1 : inl 7 o< inr 0.
Proof. constructor. Qed.
```

We can now define abbreviations. For instance, the finite ordinals are represented by terms built with the constructor `inl`, and the first infinite ordinal ω by the term `(inr 0)`.

```
Definition fin (i:nat) : t := inl i.
Coercion fin : nat >-> t.

Notation "'omega'" := (inr  0:t).
```

```
Example ex1' : fin 7 o< omega.
Proof. constructor. Qed.

Lemma lt_omega alpha :
     alpha o< omega <-> exists n:nat,  alpha = fin n.
(* ... *)
```

3.6 Limits and Successors

Let us look again at our implementation of $\omega + \omega$. We can distinguish between the three kinds of ordinals seen in Fig 3.1:

- The least ordinal, `(inl 0)`, also written `(fin 0)`.
- The limit ordinal ω.
- The successor ordinals, either of the form `(inl (S` i`))` or `(inr (S` i`))`

3.6.1 Definitions

It would be interesting to specify at the most generic level, what is a zero, a successor or a limit ordinal. Let $<$ be a strict order on a type A.

- A *least* element is a minorant (in the large sense) of the full set on A,
- y is a *successor* of x if $x < y$ and there is no element between x and y. We will also say that x is a *predecessor* of y.
- x is a *limit* if x is not a least element, and for any y such that $yo < x$, there exists some z such that $y < z < x$.

The following definitions are in Library Prelude.MoreOrders.

```
Section A_given.
  Variables (A : Type)  (lt: relation A).

Local Infix "<" := lt.
Local Infix "<=" := (clos_refl _ lt).

Definition Least {sto : StrictOrder lt} (x : A):=
  forall y,  x <= y.

Definition Successor {sto : StrictOrder lt} (y x : A):=
```

```
  x < y /\ (forall z,  x < z ->  z <  y -> False).

Definition Limit {sto : StrictOrder lt}  (x:A)  :=
  (exists w:A,  w < x) /\
  (forall y:A, y < x -> exists z:A, y < z /\ z < x).
```

Exercise 3.1 Prove, that, in any ordinal notation system, that every ordinal has at most one predecessor, and at most one successor.

Exercise 3.2 Prove, that, in any ordinal notation system, that if β is a successor of α, then for any γ, $\gamma < \beta$ implies $\gamma \leq \alpha$.

3.6.2 Limits and Successors in $\omega + \omega$

Using the definitions above, we can prove the following lemma:

From Module OrdinalNotations.ON_Omega_plus_omega

```
Lemma limit_iff (alpha : t) : Limit alpha <-> alpha = omega.
```

Regarding successors, let us introduce the following definition:

```
Definition succ (alpha : t) :=
  match alpha with
    inl n => inl (S n)
  | inr n => inr (S n)
  end.
```

```
Lemma Successor_correct alpha beta : Successor beta alpha <->
                                     beta = succ alpha.
```

We can also check whether an ordinal is a successor by a simple pattern matching:

```
Definition succb (alpha: t) : bool := match alpha with
                                  | inr (S  _) | inl (S _) => true
                                  | _ => false
                                  end.

Lemma succb_correct (alpha : t) :
    succb alpha <->  exists beta: t, alpha = succ beta.
```

Finally, the nature of any ordinal is decidable(inside this notation system) :
From Module OrdinalNotations.Generic

```
  Definition ZeroLimitSucc_dec {A:Type}{lt: relation A}
           {compare : A -> A -> comparison}
           {on : ON lt compare} :=
  forall alpha,
    {Least alpha} +
```

```
    {Limit alpha} +
    {beta: A | Successor alpha beta}.
(* ... *)
```

From Module OrdinalNotations.ON_Omega_plus_omega

```
Definition Zero_limit_succ_dec : ZeroLimitSucc_dec.
```

3.7 The Ordinal ω^2

The ordinal ω^2 (also called $\phi_0(2)$, see Chap. 7), is represented by the lexicographically ordered cartesian product $\mathbb{N} \times \mathbb{N}$.

From Module ON_Omega2

```
Definition t := (nat * nat)%type.

Definition lt : relation t := lexico Peano.lt Peano.lt.
Definition le := clos_refl _ lt.

Infix "o<" := lt : o2_scope.
Infix "o<=" := le : o2_scope.
```

It is easy to represent finite ordinals, successors and limits.

```
Definition zero: t := (0,0).

Definition fin (n:nat) : t := (0, n).

Notation "'F'" := fin : o2_scope.

Coercion fin : nat >-> t.

Definition succ (alpha : t) := (fst alpha, S (snd alpha)).

Notation "'omega'" := (1,0) : o2_scope.

Definition limitb (alpha : t) :=
 match alpha with
 |  (S _, 0) => true
 | _ => false
end.
```

In order to build an ordinal notation out of the type `t`, we have to define a comparison function, and prove that the order `lt` is well-founded.

```
Instance lt_strorder : StrictOrder lt.
(* ... *)

Definition compare (alpha beta: t) : comparison :=
  match Nat.compare (fst alpha) (fst beta) with
```

```
    Eq => Nat.compare (snd alpha) (snd beta)
  | c => c
  end.

Lemma compare_correct alpha beta :
    CompareSpec (alpha = beta) (lt alpha beta) (lt beta alpha)
                (compare alpha beta).

Lemma lt_wf : well_founded lt.

Instance Omega2: ON lt compare.
Proof.
  split.
  - apply lt_strorder.
  - apply lt_wf.
  - apply compare_correct.
Qed.
```

3.7.1 Arithmetic of ω^2

3.7.1.1 Addition

We can define on `ON_omega2` an addition which extends the addition on `nat`.

```
Definition  plus (alpha beta : t) : t :=
  match alpha,beta with
  | (0, b), (0, b') => (0, b + b')
  | (0,0), y  => y
  | x, (0,0)  => x
  | (0, b), (S n', b') => (S n', b')
  | (S n, b), (S n', b') => (S n + S n', b')
  | (S n, b), (0, b') => (S n, b + b')
   end.

Infix "+" := plus : o2_scope.
```

Please note that this operation is not commutative:

```
Example non_commutativity_of_plus :  omega + 3 <> 3 + omega.
Proof.
  cbn.
```

```
1 subgoal (ID 237)

  ============================
(1, 3) <> omega.
```

```
 discriminate.
Qed.
```

3.7.1.2 Multiplication

The restriction of ordinal multiplication to the segment $[0, \omega^2)$ is not a total function. For instance $\omega \times \omega = \omega^2$ is outside the set of represented values. Nevertheless, we can define two operations mixing natural numbers and ordinals.

```
(** multiplication of an ordinal by a natural number *)

Definition mult_fin_r  (alpha : t) (p : nat): t :=
  match alpha, p with
 |  (0,0), _  => zero
 |  _, 0 => zero
 |  (0, n), p => (0, n * p)
 |  ( n, b),  n' => ( n *  n', b)
               end.
Infix "*" := mult_fin_r : o2_scope.

(** multiplication of  a natural number by an ordinal *)

Definition mult_fin_l (n:nat)(alpha : t) : t :=
  match n, alpha with
 |  0, _  => zero
 |  _, (0,0) => zero
 |   n , (0,n') => (0, (n*n')%nat)
 |  n, (n',p') => (n', (n * p')%nat)
 end.

Example e1 : (omega * 7 + 15) * 3 = omega * 21 + 15.
Proof. reflexivity. Qed.

Example e2 :  mult_fin_l 3 (omega * 7 + 15) = omega * 7 + 45.
Proof. reflexivity. Qed.
```

Multiplication with a finite ordinal and addition are related through the following lemma:

```
Lemma unique_decomposition alpha :
    exists! i j: nat,  alpha = omega * i + j.
```

3.7.2 A Proof of Termination using ω^2

.

Using the lemma of Sect. 3.4.2 on page 47, we can define easily a total function which merges two lists.

```
(* adapted from Pascal Manoury et al. *)


Require Import Coq.Program.Wf List.
Require Import FunInd Recdef.

Section Merge.
```

```
  Variable A: Type.

  Local Definition m (p : list A * list A) :=
    omega * length (fst p) + length (snd p).

  Function  merge  (ltb: A -> A -> bool)
          (xys: list A * list A)
          {wf (measure_lt m) xys} :
    list A :=
    match xys with
      (nil, ys) => ys
    | (xs, nil) => xs
    | (x :: xs, y :: ys) =>
      if ltb x y then x :: merge  ltb (xs, (y :: ys))
      else y :: merge  ltb ((x :: xs), ys)
    end.

  - intros; unfold m, measure_lt; cbn; destruct xs0; simpl; left; lia.
  - intros; unfold m, measure_lt; cbn; destruct ys0; simpl; right; lia.
   - auto.
  Defined.

End Merge.

Goal forall l, merge nat Nat.leb (nil, l) = l.
  intro; now rewrite merge_equation.
Qed.
```

3.7.3 Yet Another Proof of Impossibility

In Sect. 2.4.3 on page 39, we proved that there exists no variant towards `nat` (*i.e.* the ordinal ω) for proving the termination of all hydra battles. We prove now that the ordinal ω^2 is also insufficient for this purpose.

The proof we are going to develop has exactly the same structure as in Section 2.4.3. Nevertheless, the proof of technical lemmas is a little more complex, due to the structure of the lexicographic order on $\mathbb{N} \times \mathbb{N}$. Consider for instance that there exists an infinite number of ordinals between ω and $\omega \times 2$.

The detailed proof script is in the file `../theories/html/hydras.Hydra.Omega2_Small.html`.

3.7.3.1 Preliminaries

Let us assume there is a variant from `Hydra` into ω^2 for proving the termination of all hydra battles.

From Module Hydra.Omega2_Small

```
 Section Impossibility_Proof.

Variable m : Hydra -> ON_Omega2.t.

Context (Hvar : Hvariant lt_wf free m).
```

Let us follow the same pattern as in Sect. 2.4.3. First, we define an injection from type `t` into `Hydra`, by associating to each ordinal $\omega \times i + j = (i, j)$ the hydra with i branches of length 2 and j branches of length 1.

From Module Hydra.Omega2_Small

```
Let iota (p: ON_Omega2.t) :=
    node (hcons_mult (hyd1 head) (fst p)
                     (hcons_mult head (snd p) hnil)).
```

For instance, Figure 3.3 shows the hydra associated to the ordinal $(3, 5)$, a.k.a. $\omega \times 3 + 5$.

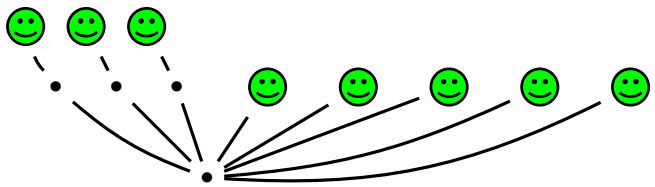

Figure 3.3: The hydra $\iota(\omega \times 3 + 5)$

Like in Sect. 2.4.3, we build a hydra out of the range of `iota` (represented in Fig. 3.4).

Figure 3.4
The hydra `big_h`.

```
  Let big_h := hyd1 (hyd2 head head).
```

In a second step, we build a "smaller" hydra.

```
  Let small_h := iota (m big_h).
```

Like in Sect. 2.4.3, we prove the double inequality `m big_h o<= m small_h o< m big_h`, which is impossible.

3.7.3.2 Proof of the Inequality `m small_h o< m big_h`

In order to prove the inequality `m_lt: m small_h o< m big_h`, it suffices to build a battle transforming `big_h` into `small_h`.

First we prove that `small_h` is reachable from `big_h` in one or two steps. Let us decompose `m big_h` as (i, j). If $j = 0$, then one round suffices to transform

`big_h` into $\iota(i,j)$. If $j > 0$, then a first round transforms `big_h` into $\iota(i+1,0)$ and a second round into $\iota(i,j)$. So, we have the following result.

```
Lemma big_to_small: big_h -+-> small_h.
```

Since m is a variant, we infer the following inequality:

```
Corollary m_lt : m small_h o< m big_h.
```

3.7.3.3 Proof of the Inequality `m big_h o<= m small_h`

The proof of the inequality `m big_h o<= m small_h` is quite more complex than in Sect 2.4.3. If we consider any ordinal $\alpha = (i,j)$, where $i > 0$, there exists an infinite number of ordinals stricly less than α, and there exists an infinite number of battles that start from $\iota(\alpha)$. Indeed, at any configuration $\iota(k,0)$, where $k > 0$, the hydra can freely choose any replication number. Intuitively, the measure of such a hydra must be large enough for taking into account all the possible battles issued from that hydra. Let us now give more technical details.

- The proof of the lemma `m_ge : m big_h o<= m small_h` uses well-founded induction on `big_h`.
- For any pair p, we have to distinguish between three cases, according to the value of p's components.
  - $p = (0,0)$
  - $p = (i,0)$, where $i > 0$: p corresponds to a limit ordinal
  - $p = (i,j)$, where $j > 0$: p is the successor of $(i, j-1)$.

Let us define the notion of elementary "step" of decreasing sequences in `t`

```
Inductive step : t -> t -> Prop :=
| succ_step : forall i j,  step (i, S j) (i, j)
| limit_step : forall i j, step (S i, 0) (i, j).
```

The following lemma establishes a correspondance between the relation `step` and hydra battles.

```
Lemma step_to_battle : forall p q, step p q -> iota p -+-> iota q.
```

Thus, starting from any inequality $q < p$ on type `t`, we can build by *transfinite induction* (*i.e.* well-founded) over `p` a battle that transforms the hydra $\iota(p)$ into $\iota(q)$.

From Module Hydra.Omega2_Small

```
Lemma m_ge : forall p : t,   p o<= m (iota p).
Proof.
  unfold small_h; pattern (m big_h) .
   apply  well_founded_induction with (R := lt) (1:= lt_wf).
  intro p ; pattern p;
    apply  well_founded_induction with
                (R := lt2) (1:= wf_lexico lt_wf lt_wf);
     intros (i,j) IHij.
```

```
  i, j : nat
  IHij : forall y : t, y o< (i, j) -> y o<= m (iota y)
  ============================
  (i, j) o<= m (iota (i, j))
```

Then we have three cases to consider, according to the values of i and j.

- If $p = (0, 0)$ then obviously, $\iota(p) \geq p = (0, 0)$
- If $p = (i + 1, 0)$ for some $i \in \mathbb{N}$, we remark that p is strictly greater than any pair (i, j), where j is any natural number.

  Applying the battle rules, for any j, we have $\iota(i + 1, j)$ `-1->` $\iota(i, j)$, thus $m(\iota(p)) > m(\iota(i, j)$ since m is assumed to be a variant.

  Applying the induction hypothesis, we get the inequality $m(\iota(i, j)) \geq (i, j)$ for any j.

  Thus, $m(\iota(p)) > (i, j)$ for any j. Applying the lemma `limit_is_lub`, we get the inequality $m(\iota(i + 1, 0)) \geq (i + 1, 0)$
- If $p = (i, j + 1)$ with $j \in \mathbb{N}$, we have $\iota(p)$ `-1->` $\iota(i, j)$, hence $m(\iota(p)) > m(\iota(i, j)) \geq (i, j)$, thus $m(\iota(p)) \geq (i, j + 1) = p$

```
  (* ... *)
Qed.
```

3.7.3.4 End of the Proof

Since $<$ is a strict order (irreflexive and transitive) on `nat*nat`, we infer that there is no variant for termination on the lexicographic square of $(\mathbb{N}, <)$.

1. From `m_lt`, we infer the strict inequality `m small_h o< m big_h`
2. from `m_ge`, we get `m big_h o<= m (iota (m big_h)) = m small_h`

From Module Hydra.Omega2_Small

```
Theorem Impossible : False.
Proof.
  destruct (StrictOrder_Irreflexive  (m big_h)).
  apply le2_lt2_trans with (m small_h).
  -  unfold small_h;  apply m_ge.
  -  apply m_lt.
Qed.

End Impossibility_Proof.
```

Exercise 3.3 Prove that there exists no variant m from `Hydra` into ω^2 for proving the termination of all *standard* battles.

Remark 3.2 In Chapter 5, we prove a generalization of the impossibility lemmas of Sect. 2.4.3 and this section, with the same proof structure, but with much more complex technical details.

Exercise 3.4 Write *direct* proofs (i.e., without applying the result and tools of Chap. 5) that the following data structures are too simple for defining a variant for any hydra battle.

- ω^n : the set of all n-uples of natural numbers, ordered by lexicographic ordering
- ω^ω: the set of all decreasing sequences (with respect to $\leq$) of natural numbers, ordered by lexicographic ordering on lists.

  For instance, the following inequality holds:

$$\langle 4,3,3,3,3,3,3,2,2,2\rangle < \langle 4,4,2\rangle$$

3.8 A Notation for Finite Ordinals

Let n be some natural number. The segment associated with n is the interval $[0,n) = \{0,1,\ldots,n-1\}$. One may represent the ordinal n by a sigma type.

From Module OrdinalNotations.ON_Finite

```
Coercion is_true: bool >-> Sortclass.

Definition t (n:nat) := {i:nat | Nat.ltb i n}.
```

The order on type (`t` n) is defined through the projection on `nat`.

```
Definition lt {n:nat} : relation (t n) :=
  fun alpha beta => Nat.ltb ( proj1_sig alpha) (proj1_sig beta).
```

For instance, let us build two elements of the segment $[0,7)$, *i.e.* two inhabitants of type (`t 7`), and prove a simple inequality (see Fig. 3.5).

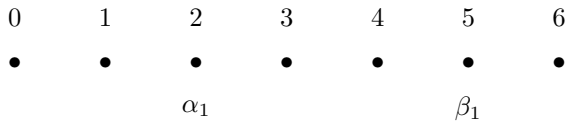

Figure 3.5: The segment $\mathbb{O}_7$

```
Program Example alpha1 : t 7 := 2.

Program Example beta1 : t 7 := 5.

Example i1 : lt  alpha1 beta1.
Proof. now compute. Qed.
```

Note that the type (`t 0`) is empty, and that, for any natural number n, n does not belong to (`t` n).

```
Lemma t0_empty (alpha: t 0): False.
Proof.
  destruct alpha.
  destruct x; cbn in i; discriminate.
Qed.


Program Definition bad : t 10 := 10.
Next Obligation.
  compute.
```

```
1 subgoal (ID 162)

  ============================
  false = true
```

```
Abort.
```

Note also that attempting to compare a term of type (`t` n) with a term of type (`t` p) leads to an error if n and p are not convertible.

```

Program Example gamma1 : t 8 := 7.

Fail Goal lt alpha1 gamma1.
```

```
 The command has indeed failed with message:
The term "gamma1" has type "t 8" while it is expected to have type "t 7".
```

In order to build an instance of `OrdinalNotation`, we define a comparison function, by delegation to standard library's `Nat.compare`, and prove its correction.

```
Definition compare {n:nat} (alpha beta : t n) :=
  Nat.compare (proj1_sig alpha) (proj1_sig beta).

Lemma compare_correct {n} (alpha beta : t n) :
  CompareSpec (alpha = beta) (lt alpha beta) (lt beta alpha)
              (compare alpha beta).
```

Remark 3.3 The proof of `compare_correct` uses a well-known pattern of Coq. Let us consider the following subgoal.

```
 1 subgoal (ID 110)

  n, x0 : nat
  i, i0 : x0 <? S n
```

```
  ============================
  exist (fun i1 : nat => i1 <=? n) x0 i =
  exist (fun i1 : nat => i1 <=? n) x0 i0
```

Applying the tactic `f_equal` generates a simpler subgoal.

```
1 subgoal (ID 112)

  n, x0 : nat
  i, i0 : x0 <? S n
  ============================
  i = i0
```

We have now to prove that there exists at most one proof of (`Nat.ltb x0 (S n)`). This is not obvious, but a consequence of the following lemma of library Coq.Logic.Eqdep_dec.

```
eq_proofs_unicity_on :
forall (A : Type) (x : A),
(forall y : A, x = y \/ x <> y) ->
forall (y : A) (p1 p2 : x = y), p1 = p2
```

Thus unicity of proofs of `Nat.ltb x0 (S n)` comes from the decidability of equality on type `bool`. This is why we used the boolean function `Nat.ltb` instead of the inductive predicate `Nat.lt` in the definition of type `t` n (see page 59). For more information about this pattern, please look at the numerous mailing lists and FAQs on Coq).

Applying lemmas of the libraries `Coq.Wellfounded.Inverse_Image`, `Coq.Wellfounded.Inclusion`, and `Coq.Arith.Wf_nat`, we prove that our relation `lt` is well founded.

```
Lemma lt_wf (n:nat) : well_founded (@lt n).
```

Now we can build our instance of `OrdinalNotation`.

```
Global Instance sto n : StrictOrder (@lt n).

Global Instance FinOrd (n:nat) : OrdinalNotation (sto n) compare.
Proof.
  split.
  - apply compare_correct.
  - apply lt_wf.
Qed.
```

Remark 3.4 It is important to keep in mind that the integer n is not an "element" of `FinOrd` n. In set-theoretic presentations of ordinals, the set associated with the ordinal n is $\{0, 1, \ldots, n-1\}$. In our formalization, the interpretation of an ordinal as a set is realized by the following definition (in OrdinalNotations.Generic).

```
Definition bigO `{nA : @OrdinalNotation A ltA stoA compareA}
           (a: A) : Ensemble A :=
  fun x: A => ltA x a.
```

Remark 3.5 There is no interesting arihmetic on finite ordinals, since functions like successor, addition, etc., cannot be represented in Coq as *total* functions.

Remark 3.6 Finite ordinals are also formalized in MathComp [MT]. See also Adam Chlipala's *CPDT* [Chl11] for a thorough study of the use of dependent types.

3.9 Comparing two Ordinal Notations

It is sometimes useful to compare two ordinal notations with respect to expressive power (the segment of ordinals they represent).

The following class specifies a strict inclusion of segments. The notation `OA` describes a segment $[0, \alpha($, and `OB` is a larger segment (which contains a notation for α, whilst α is not represented in `OA`). We require also that the comparison functions of the two notation systems are compatible.

If `OB` is presumed to be correct, the we can consider that `OA` "inherits" its correcteness from the bigger notation system `OB`.

```
Class  SubON
       `(OA : @ON A ltA  compareA)
       `(OB : @ON B ltB  compareB)
       (alpha :  B)
       (iota : A -> B):=
  {
  SubON_compare: forall x y : A,  compareB (iota x) (iota y) =
                                  compareA x y;
  SubON_incl : forall x, ltB (iota x) alpha;
  SubON_onto : forall y, ltB y alpha  -> exists x:A, iota x = y}.
```

For instance, we prove that `Omega` is a sub-notation of `Omega_plus_Omega` (with ω as the first "new" ordinal, and `fin` as the injection).

```
Instance Incl : SubON Omega Omega_plus_Omega omega fin.
```

We can also show that, if $i < j$, then the segment $[0, i)$ is a "sub-segment" of $[0, j)$. Since the terms $(t\ i)$ and $(t\ j)$ are not convertible, we consider a "cast" function ι from $(t\ i)$ into $(t\ j)$, and prove that this function is a monotonous bijection from $(t\ i)$ to the segment $[0, i)$ of $(t\ j)$.

We are now able to build an instance of `SubON`.

From Module OrdinalNotations.ON_Finite

```
Section Inclusion_ij.

  Variables i j : nat.
```

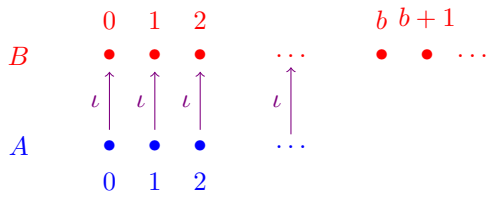

Figure 3.6: A is a sub-segment of B

```
Hypothesis Hij : (i < j)%nat.

Remark Ltb_ij : Nat.ltb i j.

Program Definition iota_ij  (alpha: t i) : t j :=  alpha.

 Let b : t j := exist _ i Ltb_ij.

 Global Instance F_incl_ij  : SubON  (FinOrd i) (FinOrd j) b iota_ij.
(* ... *)

End Inclusion_ij.
```

Exercise 3.5 Prove that `Omega_plus_Omega` cannot be a sub-notation of `Omega`.

Project 3.1 Adapt the definition of `Hvariant` (Sect. 2.4.2) in order to have an ordinal notation as argument. Prove that if O_A is a sub-notation of O_B, then any variant defined on O_A can be automatically transformed into a variant on O_B.

3.10 Comparing an Ordinal Notation with Schütte's Model

Finally, it may be interesting to compare an ordinal notation with the more theoretical model from Schütte (well, at least with our formalization of that model). This would be a relative proof of correctenss of the considered ordinal notation.

The following class specifies that a notation `OA` describes a segment $[0, \alpha)$, where α is a countable ordinal *à la* Schütte.

```
Class ON_correct `(alpha : Schutte_basics.Ord)
     `(OA : @ON A ltA  compareA)
      (iota : A -> Schutte_basics.Ord) :=
  { ON_correct_inj : forall a, Schutte_basics.lt (iota a) alpha;
    ON_correct_onto : forall beta, Schutte_basics.lt beta alpha ->
                                exists b, iota b = beta;
    On_compare_spec : forall a b:A,
        match compareA a b with
          Datatypes.Lt => Schutte_basics.lt (iota a) (iota b)
```

```
        | Datatypes.Eq => iota a = iota b
        | Datatypes.Gt => Schutte_basics.lt (iota b) (iota a)
        end}.
```

For instance, the following theorem tells that `Epsilon0`, our notation system for the segment $[0, \epsilon 0)$ is a correct implementation of the theoretically defined ordinal ϵ_0 (see chapter 7 for more details).

```
Instance Epsilon0_correct :
  ON_correct epsilon0 Epsilon0  (fun alpha => inject (cnf alpha)).
```

Project 3.2 When you have read Chapter 7, prove that the sum of two ordinal notations `ON_plus` implements the addition of ordinals.

3.11 Isomorphism of Ordinal Notations

In some cases we want to show that two notation systems describe the same segment (for instance $[0, 3 + \omega)$ and $[0, \omega( )$. For this purpose, one may prove that the two notation systems are order-isomorphic.

```
Class  ON_Iso
       `(OA : @ON A ltA compareA)
       `(OB : @ON B ltB  compareB)
       (f : A -> B)
       (g : B -> A):=
  {
  iso_compare: forall x y : A,
      compareB (f x) (f y) = compareA x y;
  iso_inv1 : forall a, g (f a)= a;
  iso_inv2 : forall b, f (g b) = b}.
```

Exercise 3.6 Let i be some natural number. Prove that the notation systems `Omega` and `ON_plus (OrdFin i) Omega` are isomorphic.

Note: *This property reflects the equality $i + \omega = \omega$, that we will prove in larger notation systems, as well as in Schütte's model.*

This exercise is partially solved for $i = 3$ (in OrdinalNotations.Example_3PlusOmega).

Project 3.3 Define in Coq the product of two ordinal notations N_A and N_B. If A [resp. B] is the underlying type of N_A [resp. N_B], the product `ON_mult` N_A N_B is implemented over the cartesian product $B \times A$ (with the lexicographic ordering).

For instance, the elements of the product `ON_mult Omega (FinOrd 3)` are ordered as follows.

$$(0,0),(0,1),(0,2),(0,3),(0,4),\ldots,(1,0),(1,1),(1,2),\ldots,(2,0),(2,1),(2,2),\ldots$$

Note that the elements of (`ON_mult (FinOrd 3) Omega`) are differently ordered (without limit ordinals):

$$(0,0),(1,0),(2,0),(0,1),(1,1),(2,1),(0,2),(1,2),(2,2),(0,3),\ldots$$

Prove that `ON_mult (FinOrd` i`) Omega` is isomorphic to `Omega` whilst `Omega` is a sub-notation of `ON_mult Omega (FinOrd` i`)`, for any strictly positive i.

Project 3.4 Consider two isomorphic ordinal notations `OA` and `OB`. Prove that, if `OA` [resp. `OB`] is a correct implementation of α [resp. β], then $\alpha = \beta$.

Project 3.5 Add to the class `ON` the requirement that for any α it is decidable whether α is 0, a successor or a limit ordinal.

Hint: Beware of the instances associated with sum and product of notations! You may consider additional fields to make the sum and product of notations "compositional".

Project 3.6 Reconsider the class `ON`, with an equivalence instead of Leibniz equality.

3.12 Other Ordinal Notations

Project 3.7 The directory `theories/OmegaOmega` contains an ad-hoc formalization of ω^ω, contributed by Pascal Manoury. Every ordinal α is represented by a list l whose elements are the coefficients of ω in the Cantor normal form of α (in reverse order). For instance, the ordinal $\omega^8 \times 5 + \omega^6 \times 8 + \omega^2 \times 10 + \omega + 7$ is represented by the list `[5;0;8;0;0.0;10,1,7]`.

Develop this representation and compare it with the other ordinal notations.

Project 3.8 Let N_A be a notation system for ordinals strictly less than α, with the strict order $(A, <_A)$. Please build the notation system `ON_Expl` N_A, on the type of multisets of elements of A (or, if preferred, the type of non-increasing finite sequences on A, provided with the lexicographic ordering on lists).

For instance, let us take $N_A =$ `Omega`, and take $\alpha = \langle 4,4,2,1,0\rangle$, $\beta = \langle 4,3,3,3,3,3,2\rangle$, and $\gamma = \langle 5\rangle$. Then $\beta < \alpha < \gamma$.

In contrast the list $\langle 5,6,3,3\rangle$ is not non-increasing (*i.e.* sorted w.r.t. $\geq$), so it is not to be considered.

Note that if the notation N_A implements the ordinal α, the new notation ω^{N_A} must implement the ordinal $\phi_0(\alpha)$, a.k.a. ω^α (see chapter 7)

Remark 3.7 The set of ordinal terms in Cantor normal form (see Chap. 4) and in Veblen normal form (see Gamma0.Gamma0) are shown to be ordinal notation systems, but there is a lot of work to be done in order to unify ad-hoc definitions and proofs which were written before the definition of the `ON` type class.

Chapter 4

A Proof of Termination, using Ordinals below ϵ_0

In this chapter, we adapt to Coq the well-known [KP82] proof that Hercules eventually wins every battle, whichever the strategy of each player. In other words, we present a formal and self contained proof of termination of all [free] hydra battles. First, we take from Manolios and Vroon [MV05] a representation of the ordinal ϵ_0 as terms in Cantor normal form. Then, we define a variant for hydra battles as a measure that maps any hydra to some ordinal strictly less than ϵ_0.

4.1 The Ordinal ϵ_0

4.1.1 Cantor Normal Form

The ordinal ϵ_0 is the least ordinal number that satisfies the equation $\alpha = \omega^\alpha$, where ω is the least infinite ordinal. Thus, we can consider ϵ_0 as an *infinite* ω-tower. Nevertheless, any ordinal strictly less that ϵ_0 can be finitely represented by a unique *Cantor normal form*, that is, an expression which is either the ordinal 0 or a sum $\omega^{\alpha_1} \times n_1 + \omega^{\alpha_2} \times n_2 + \cdots + \omega^{\alpha_p} \times n_p$ where all the α_i are ordinals in Cantor normal form, $\alpha_1 > \alpha_2 > \alpha_p$, and all the n_i are positive integers.

An example of Cantor normal form is displayed in Fig 4.1: Note that any ordinal of the form $\omega^0 \times i + 0$ is just written i.

$$\omega^{(\omega^\omega + \omega^2 \times 8 + \omega)} + \omega^\omega + \omega^4 + 6$$

Figure 4.1: An ordinal in Cantor normal form

In the rest of this section, we define an inductive type for representing in `Coq` all the ordinals strictly less than ϵ_0, then extend some arithmetic operations to this type, and finally prove that our representation fits well with the expected mathematical properties: the order we define is a well order, and the

decomposition into Cantor normal form is consistent with the implementation of the arithmetic operations of exponentiation of base ω and addition.

Remark Unless explicitly mentionned, the term "ordinal" will be used instead of "ordinal strictly less than ϵ_0" (except in Chapter 7 where it stands for "countable ordinal").

4.1.2 A Data Type for Ordinals in Cantor Normal Form

Let us define an inductive type whose constructors are respectively associated with the ways to build Cantor normal forms:

- the ordinal 0
- the construction $(\alpha, n, \beta) \mapsto \omega^\alpha \times (n+1) + \beta \quad (n \in \mathbb{N})$

From Module Epsilon0.T1

```
Inductive T1 : Set  :=
| zero : T1
| ocons : T1 -> nat -> T1 -> T1.
```

Remark

The name `T1` we gave to this data-type is proper to this development and refers to a hierarchy of ordinal notations. For instance, in [CC06], the following type is used to represent ordinals strictly less than Γ_0, in Veblen normal form (see also [Sch77]).

Please look also at the library Gamma0.T2.html.

```
Inductive T2 : Set :=
  zero : T2
| gcons : T2 -> T2 -> nat -> T2 -> T2.
```

4.1.2.1 Example

For instance, the ordinal $\omega^\omega + \omega^3 \times 5 + 2$ is represented by the following term:

```
Example alpha_0 : T1 :=
  ocons (ocons (ocons zero 0 zero)
               0
               zero)
        0
       (ocons (ocons zero 2 zero)
              4
              (ocons zero 1 zero)).
```

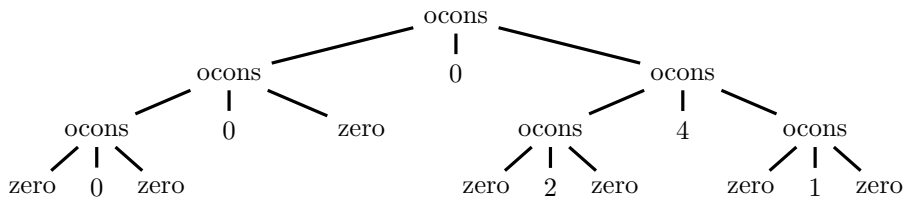

Figure 4.2: The tree-like representation of the ordinal $\omega^\omega + \omega^3 \times 5 + 2$

4.1.2.1.1 Remark For simplicity's sake, we chosed to forbid expressions of the form $\omega^\alpha \times 0 + \beta$. Thus, the contruction (`ocons` α n β) is intented to represent the ordinal $\omega^\alpha \times (n+1) + \beta$ and not $\omega^\alpha \times n + \beta$. In a future version, we should replace the type `nat` with `positive` in `T1`'s definition. But this replacement would take a lot of time …

4.1.3 Abbreviations

Some abbreviations may help to write more concisely complex ordinal terms.

4.1.3.1 Finite Ordinals

For representing finite ordinals, *i.e.* natural numbers, we first introduce a notation for terms of the form $n+1$, then define a coercion from type `nat` into `T1`.

```
Notation "'FS' n" :=
     (ocons zero n zero) (at level 10) : t1_scope.
```

```
Definition fin (n:nat) : T1 :=
    match n with 0 => zero | S p => FS p end.

Coercion fin  : nat >-> T1.

Example ten : T1 := 10.
```

4.1.3.2 The Ordinal ω

Since ω's Cantor normal form is i.e. $\omega^{\omega^0} \times 1 + 0$, we can define the following abbreviation:

```
Notation omega := (ocons (ocons zero 0 zero) 0 zero): t1_scope.
```

Note that `omega` is not an identifier, thus any tactic like `unfold omega` would fail.

4.1.3.3 The Ordinal ω^α, a.k.a. $\phi_0(\alpha)$

We provide also a notation for ordinals of the form ω^α.

```
Notation "'phi0' alpha" := (ocons alpha 0 zero) (at level 29) : t1_scope.
```

Remark 4.1 The name ϕ_0 comes from ordinal numbers theory. In [Sch77], Schütte defines ϕ_0 as the ordering (*i.e.* enumerating) function of the set of *additive principal ordinals i.e.* strictly positive ordinals α that verify $\forall \beta < \alpha, \beta + \alpha = \alpha$. For Schütte, ω^α is just a notation for $\phi_0(\alpha)$. See also Chapter 7 of this document.

4.1.3.4 The Hierarchy of ω-towers:

The ordinal ϵ_0, although not represented by a finite term in Cantor normal form, is approximed by the sequence of ω-towers (see also Sect 7.6.3 on page 139).

From Module Epsilon0.T1

```
Fixpoint omega_tower (height:nat) : T1 :=
 match height with
 | 0 =>  1
 | S h => phi0 (omega_tower h)
 end.
```

For instance, Figure 4.3 represents the ordinal returned by the evaluation of the term `omega_tower 7`.

$$\omega^{\omega^{\omega^{\omega^{\omega^{\omega^{\omega}}}}}}$$

Figure 4.3: The ω-tower of height 7

4.1.4 Comparison between Ordinal Terms

In order to compare two terms of type `T1`, we define a recursive function `compare` that maps two ordinals α and β to a value of type `comparison`. This type is defined in Coq's standard library `Init.Datatypes` and contains three constructors: `Lt` (less than), `Eq` (equal), and `Gt` (greater than).

From Module Epsilon0.T1

```
Fixpoint compare (alpha alpha':T1):comparison :=
  match alpha, alpha' with
    zero, zero => Eq
  | zero, ocons a' n' b' => Lt
  | _   , zero => Gt
  | (ocons a n b),(ocons a' n' b') =>
      (match compare a a' with
          | Lt => Lt
          | Gt => Gt
```

```
          | Eq => (match lt_eq_lt_dec n n'
                    with
                        inleft  (left _) => Lt
                      | inright _ => Gt
                      |   _ => compare b b'
                    end)
       end)
  end.
```

It is now easy to define the boolean predicate `lt_b` α β: " α is strictly less than β ". By coercion to sort `Prop` we define also the predicate `lt`.

From Module Epsilon0.T1

```
Definition lt_b alpha beta : bool :=
  match compare alpha beta with
      Lt => true
    | _ => false
  end.

Definition lt alpha beta : Prop := lt_b alpha beta.
```

Please note that this definition of `lt` makes it easy to write proofs by reflection, as shown by the following exampgles.

```
Example E1 : lt (ocons omega 56 zero) (tower 3).
Proof. reflexivity. Qed.

Example E2 : ~ lt (tower 3) (tower 3).
Proof.  discriminate.  Qed.
```

The following lemmas establish relations between `compare`, the predicate `lt` and Leibniz equality `eq`.

From Module Epsilon0.T1

```
Lemma compare_refl : forall alpha, compare alpha alpha =  Eq.
```

```
Lemma compare_reflect : forall alpha beta,
    match compare alpha beta with
    |   Lt => lt alpha  beta
    |   Eq => alpha = beta
    |   Gt => lt beta  alpha
    end.
```

We prove also that the relation `lt` is a strict total order.

From Module Epsilon0.T1

```
Theorem lt_irrefl (alpha: T1):  ~ lt alpha alpha.

Theorem lt_trans (alpha beta gamma : T1) :
  lt alpha  beta -> lt beta gamma -> lt alpha gamma.
```

```
Definition lt_eq_lt_dec  :
   forall alpha beta : T1,
          {lt alpha  beta} + {alpha = beta} + {lt beta alpha}.
```

Note that the order `lt` is not reflected in the structure (size and/or height) of the terms of `T1`. For instance the ordinal of Fig 4.1 is strictly less than the structurally simpler $\omega^{\omega^{\omega}} \times 2$.

4.1.4.1 A Predicate for Characterizing Normal Forms

Our data-type `T1` allows us to write expressions that are not properly in Cantor normal form as specified in Section 4.1. For instance, consider the following term of type `T1`.

```
Example bad_term  : T1 := ocons 1 1 (ocons omega 2 zero).
```

This term would have been written $\omega^1 \times 2 + \omega^{\omega} \times 3$ in the usual mathematical notation. We note that the exponents of ω are not in the right (strictly decreasing) order.

With the help of the order `lt` on `T1`, we are now able to characterize the set of all well-formed ordinal terms:

From Module Epsilon0.T1

```
Fixpoint nf_b (alpha : T1) : bool :=
  match alpha with
    | zero => true
    | ocons a n zero => nf_b a
    | ocons a n ((ocons a' n' b') as b) =>
      (nf_b a && nf_b b && lt_b a' a)%bool
  end.

Definition nf alpha: Prop := nf_b alpha.
```

```
Compute nf_b alpha_0.
```

```
  = true
    : bool
```

```
Compute nf_b bad_term.
```

```
  = false
    : bool
```

4.1.5 Making Normality Implicit

We would like to get rid of terms of type `T1` which are not in Cantor normal form. A simple way to do this is to consider statements of the form `forall alpha: T1, nf alpha -> P alpha`, where P is a predicate over type `T1`, like in the following lemma [1].

[1] Ordinal addition is formally defined a little later (page 4.1.7.2)

```
Lemma plus_is_zero alpha beta :
  nf alpha -> nf beta ->
  alpha + beta  = zero -> alpha = zero /\  beta = zero.
```

But this style leads to clumsy statements, and generates too many sub-goals in interactive proofs (although often solved with `auto` or `eauto`).

One may encapsulate conditions of the form `(nf` α`)` in the most used predicates. For instance, we introduce the restriction of `lt` to terms in normal form, and provide a handy notation for this restriction.

From Module Ordinals.Prelude.Restriction

```
Definition restrict {A:Type}(E: Ensemble A)(R: relation A) :=
    fun a b => E a /\ R a b /\ E b.
```

From Module Epsilon0.T1

```
Definition LT := restrict nf lt.
Infix "t1<" := LT : t1_scope.

Definition LE := restrict nf le.
Infix "t1<=" := LE : t1_scope.
```

For instance, in the following lemma, the condition that α is in normal form is included in the condition $\alpha < 1$.

```
Lemma LT_one : forall alpha, alpha t1< one -> alpha = zero.
```

4.1.5.1 A Sigma Type for ϵ_0

As we noticed in Sect. 4.1.4.1, the type `T1` is not a correct ordinal notation, since it contains terms that are not in Cantor normal form. In certain contexts (for instance in Sections 6.2.4, 6.3, and 6.4), we need to define total recursive functions on well-formed ordinal terms less than ϵ_0, using the `Equations` plug-in [SM19]. In order to define a type whose inhabitants represent just ordinals, we build a type gathering a term of type `T1` and a proof that this term is in normal form.

From Module Epsilon0.E0

```
Class E0 : Type := t1_2o {cnf : T1; cnf_ok : nf cnf}.
```

Many constructs : types, predicates, functions, notations, etc., on type `T1` are adapted to `E0`.

First, we declare a notation scope for `E0`.

```
Declare Scope E0_scope.
Delimit Scope E0_scope with e0.
Open Scope E0_scope.
```

Then we redefine the predicates of comparison.

```
Definition Lt (alpha beta : E0) := T1.LT (@cnf alpha) (@cnf beta).
Definition Le (alpha beta : E0) := T1.LE (@cnf alpha) (@cnf beta).

Infix "o<" := Lt : E0_scope.
Infix "o<=" := Le : E0_scope.
```

Equality in `E0` is just Leibniz equality. Note that, since `nf` is defined by a Boolean function, for any term α : `T1`, there exists at most one proof of `nf` α, thus two ordinals of type `E0` are equal if and only iff their projection to `T1` are equal (see also Sect. 3.3 on page 61).

```
Require Import Logic.Eqdep_dec.

Lemma nf_proof_unicity :
  forall (alpha:T1) (H H': nf alpha), H = H'.

Lemma E0_eq_iff alpha beta : alpha = beta <-> cnf alpha = cnf beta.
```

Exercise 4.1 In earlier versions of this development, the predicate `nf` was defined inductively, with various constructors describing all possible cases.

1. Please give such a definition, in a dedicated module.
2. Prove the logical equivalence between your definition and ours.
3. Define a variant of the type `E0` (with your definition of `nf`).
4. Can you still prove a lemma like `E0_eq_iff` ? With the help of an axiom from some module of the standard library ?

For upgrading constants and fonctions of `T1`, we have to prove that the term they build is in normal form. For instance, let us represent the ordinals 0 and ω as instances of the class `E0`.

```
Instance Zero : E0.
Proof.
  now exists T1.zero.
Defined.

Instance _Omega : E0.
Proof.  now exists omega%t1.
Defined.

Notation "'omega'"  := _Omega : E0_scope.
```

4.1.6 Syntactic Definition of Limit and Successor Ordinals

Pattern matching and structural recursion allow us to define the notions of successor and limit ordinal with the help of boolean functions on type `T1`.

From Module Epsilon0.T1

```
  Fixpoint succb alpha :=
  match alpha with
      zero => false
    | ocons zero _ _ => true
    | ocons alpha n beta => succb beta
  end.

Fixpoint limitb alpha :=
  match alpha with
      zero => false
    | ocons zero _ _ => false
    | ocons alpha n zero => true
    | ocons alpha n beta => limitb beta
  end.
```

```
  Compute limitb omega.
```

```
  = true
     : bool
```

```
Compute succb 42.
```

```
  = true
     : bool
```

The correctness of these definitions with respect to the mathematical notions of limit and successor ordinals is established through several lemmas. For instance, Lemma `canonS_limit`, page 93, shows that if α is (syntactically) a limit ordinal, then it is the least upper bound of a strictly increasing sequence of ordinals.

The following function is very useful in constructions by cases (proofs and function definitions).

```
Definition zero_succ_limit (alpha: T1) :
    {succb alpha} + {limitb alpha} +  {alpha=zero}.
    (* ... *)
Defined.
```

4.1.7 Arithmetic on ϵ_0

4.1.7.1 Successor

The successor of any ordinal $\alpha < \epsilon_0$ is defined by structural recursion on its Cantor normal form.

From Module Epsilon0.T1

```
Fixpoint succ (alpha:T1) : T1 :=
  match alpha with
   | zero => 1
```

```
  | ocons zero n _ => ocons zero (S n) zero
  | ocons beta n gamma => ocons beta n (succ gamma)
end.
```

The following lemma establishes the connection between the function `succ` and the Boolean predicate `succb`.

```
Lemma succb_iff alpha (Halpha : nf alpha) :
 succb alpha <-> exists beta : T1, nf beta /\ alpha = succ  beta.
```

Exercise 4.2 Prove in Coq that for any ordinal $\alpha < \epsilon_0$, α is a limit if and only if for all $\beta < \alpha$, the interval $[\beta, \alpha)$ is infinite.

4.1.7.2 Addition and Multiplication

Ordinal addition and multiplication are also defined by structural recursion over the type `T1`. Please note that they use the `compare` function on some subterms of their arguments.

```
Fixpoint plus (alpha beta : T1) : T1 :=
  match alpha,beta with
 |  zero, y  => y
 |  x, zero  => x
 |  ocons a n b, ocons a' n' b' =>
    (match compare a a' with
     | Lt => ocons a' n' b'
     | Gt => (ocons a n (plus b (ocons a' n' b')))
     | Eq  => (ocons a (S(n+n')) b')
     end)
  end
where "alpha + beta" := (plus alpha beta) : t1_scope.
```

```
Fixpoint mult (alpha beta : T1) :T1 :=
  match alpha,beta with
 |  zero, y  => zero
 |  x, zero => zero
 |  ocons zero n _, ocons zero n' _ =>
                ocons zero (Peano.pred((S n) * (S n'))) zero
 |  ocons a n b, ocons zero n' b' =>
                ocons a (Peano.pred((S n) * (S n'))) b
 |  ocons a n b, ocons a' n' b' =>
     ocons (a + a') n' ((ocons a n b) * b')
 end
where  "alpha * beta" := (mult alpha beta) : t1_scope.
```

4.1.7.3 Examples

The following examples are instances of *proofs by computation*. Please note that addition and multiplication on `T1` are not commutative. Moreover, both operations fail to be strictly monotonous in their first argument.

```
Example e2 : 6 + omega = omega.
Proof. reflexivity. Qed.

Example e'2 : omega t1< omega + 6.
Proof. now compute. Qed.

Example e''2 : 6 * omega = omega.
Proof. reflexivity. Qed.

Example e'''2 : omega t1< omega * 6.
Proof. now compute. Qed.
```

```
Lemma plus_not_monotonous_l : exists alpha beta gamma : T1,
    alpha t1< beta /\ alpha + gamma = beta + gamma.
Proof.
  exists 3, 5, omega;  now  compute.
Qed.

Lemma mult_not_monotonous :  exists alpha beta gamma : T1,
      alpha t1< beta /\ alpha * gamma = beta * gamma.
Proof.
  exists 3, 5, omega; now compute.
Qed.
```

4.1.8 Pretty Printing Ordinals in Cantor Normal Form

Let us consider again the ordinal α_0 defined in section 4.1.2.1 on page 68 If we ask Coq to print its normal form, we get a hardly readable term of type `T1`.

```
Compute alpha_0.
```

```
  = ocons omega 0 (ocons (FS 2) 4 (FS 1))
    : T1
```

The following data type defines an abstract syntax for more readable ordinals terms in Cantor normal form:

```
Inductive ppT1 : Set :=
    P_fin : nat -> ppT1
  | P_add : ppT1 -> ppT1 -> ppT1
  | P_mult : ppT1 -> nat -> ppT1
  | P_exp : ppT1 -> ppT1 -> ppT1
  | P_omega : ppT1
```

The function `pp: T1 -> ppT1` converts any closed term of type `T1` into a human-readable expression. For instance, let us convert the term `alpha_0`.

```
Compute pp alpha_0.
```

```
     = (omega ^ omega + omega ^ 3 * 5 + 2)%pT1
     : ppT1
```

Project 4.1 Design (in OCaml?) a set of tools for systematically pretty printing ordinal terms in Cantor normal form.

4.1.8.1 Arithmetic on Type E0

We define an addition in type E0, using the functionT1.plus and its properties.

```
Search   (nf (_ + _)%t1).
```

```
plus_nf: forall a : T1, nf a -> forall b : T1, nf b -> nf (a + b)
```

```
Instance plus (alpha beta : E0) : E0.
Proof.
  refine (@mkord (T1.plus (@cnf alpha) (@cnf beta))_ );
    apply plus_nf; apply cnf_ok.
Defined.

Infix "+" := plus : E0_scope.

Check omega + omega.
```

```
omega + omega
     : E0
```

Exercise 4.3 Let α be an ordinal. We say that α is *infinite* iff the segment $[0, \alpha)$ is an infinite set.

1. Adapt this definition to the type E0.
2. Prove that *being infinite* is decidable on type E0.
3. Prove that α is infinite if and only if for all finite ordinal i, $i + \alpha = \alpha$.

4.2 Well-foundedness and Transfinite Induction

4.2.1 About Well-foundedness

In order to use T1 for proving termination results, we need to prove that our order < is well-founded. Then we will get *transfinite induction* for free.

The proof of well-foundedness of the strict order $<$ on Cantor normal forms is already available in the Cantor contribution by Castéran and Contejean [CC06]. That proof relies on a library on recursive path orderings written by E. Contejean. We present here a direct proof of the same result, which does not require any knowledge on r.p.o.s.

Exercise 4.4 Prove that the *total* order lt on T1 is not well-founded. **Hint:** You will have to build a counter-example with terms of type T1 which are not in Cantor normal form.

4.2.1.1 A First Attempt

It is natural to try to prove by structural induction over `T1` that every term in normal form is accessible through `LT`.

Unfortunately, it won't work. Let us consider some well-formed term $\alpha =$ `ocons` $\beta\ n\ \gamma$, and assume that β and γ are accessible through `LT`. For proving the accessibility of α, we have to consider any well formed term δ such that $\delta < \alpha$. But nothing guarantees that δ is strictly less than β nor γ, and we cannot use the induction hypotheses on β nor γ.

```
Section First_attempt.

 Lemma wf_LT : forall alpha,  nf alpha -> Acc LT alpha.
 Proof.
  induction alpha as [| beta IHbeta n gamma IHgamma].
  - split.
    inversion 1.
    destruct H2 as [H3 _];not_neg H3.
  -  split; intros delta Hdelta.
```

```
1 subgoal (ID 560)

  beta : T1
  n : nat
  gamma : T1
  IHbeta : nf beta -> Acc LT beta
  IHgamma : nf gamma -> Acc LT gamma
  H : nf (ocons beta n gamma)
  delta : T1
  Hdelta : delta t1< ocons beta n gamma
  ============================
  Acc LT delta
```

```
  Abort.
```

The problem comes from the hypothesis `Hdelta`. It does not prevent δ to be bigger that β or γ; for instance δ may be of the form `ocons` $\beta'\ p'\ \gamma'$, where $\beta' \leq \beta$ and $p' < n$. Thus, the induction hypotheses `IHbeta` and `IHgamma` are useless for finishing our proof.

4.2.1.2 Using a Stronger Inductive Predicate.

Instead of trying to prove directly that any ordinal term α in normal form is accessible through `LT`, we propose to show first that any well formed term of the form $\omega^\alpha \times (n+1) + \beta$ is accessible (which is a stronger result).

```
Let Acc_strong (alpha:T1) :=
     forall n beta,
       nf (ocons alpha n beta) -> Acc LT (ocons alpha  n beta).
```

The following lemma is an application of the strict inequality $\alpha < \omega^\alpha$. If ω^α is accessible, then α is *a fortiori* accessible.

```
Lemma Acc_strong_stronger : forall alpha,
    nf alpha -> Acc_strong  alpha -> Acc LT  alpha.
Proof.
 intros alpha H H0; apply acc_imp with (phi0 alpha).
 - repeat split; trivial.
   + now apply lt_a_phi0_a.
 -  apply H0;  now apply single_nf.
Qed.
```

Thus, it remains to prove that every ordinal strictly less than ϵ_0 is strongly accessible.

4.2.1.2.1 A helper First, we prove that, for any LT-accessible term α, α is strongly accessible too (*i.e.* any well formed term (`ocons` α n β) is accessible).

```
Lemma Acc_implies_Acc_strong :
   forall alpha, Acc LT  alpha -> Acc_strong alpha.
```

The proof is structured as an induction on α's accessibility. Let us consider an accessible term α.

```
  subgoal 1

  alpha : T1
  Aalpha : forall y : T1,  y t1< alpha -> Acc LT y
  IHalpha : forall y : T1,
       LT y alpha ->
       forall (n : nat) (beta : T1),
       nf (ocons y n beta) -> Acc LT (ocons y n beta)
  ============================
   forall (n : nat) (beta : T1),
   nf (ocons alpha n beta) -> Acc LT (ocons alpha n beta)
```

Let `n:nat` and `beta:T1` such that `ocons alpha n beta` is in normal form. We prove first that `beta` is accessible, which allows us to by well-founded induction on `beta`, and natural induction on `n`, that (`ocons alpha n beta`) is accessible. The proof, quite long, can be consulted in `../theories/html/hydras.Epsilon0.T1.html`

4.2.1.2.2 Accessibility of any well-formed ordinal term Our goal is still to prove accessibility of any well formed ordinal term. Thanks to our previous lemmas, we are almost done.

```
(* A (last) structural induction *)

Theorem nf_Acc : forall alpha, nf alpha -> Acc LT  alpha.
Proof.
 induction alpha.
-  intro; apply Acc_zero.
```

```
 -  intros; eapply Acc_implies_Acc_strong;auto.
    apply IHalpha1;eauto.
    apply nf_inv1 in H; auto.
Defined.

Corollary T1_wf : well_founded LT.
```

```
Definition transfinite_recursor :
 forall (P:T1 -> Type),
   (forall x:T1,
     (forall y:T1, nf x -> nf y ->  lt y  x -> P y) -> P x) ->
    forall alpha:T1, P alpha.
Proof.
 intros; apply well_founded_induction_type with LT.
 -  exact T1_wf;auto.
 - intros. apply X. intros; apply X0. repeat split;auto.
Defined.
```

The following tactic starts a proof by transfinite induction on any ordinal $\alpha < \epsilon_0$.

```
Ltac transfinite_induction alpha :=
  pattern alpha; apply transfinite_recursor;[ | try assumption].
```

Remark 4.2 The alternate proof of well-foundedness using Évelyne Contejean's work on recursive path ordering [Der82, CCF+10] is available in the library Epsilon0rpo.

4.2.2 An Ordinal Notation for ϵ_0

We build an instance of `ON`, and prove its correction w.r.t. Schutte's model.

```
Instance Epsilon0 : ON Lt compare.
(* ... *)
```

From Module Schutte.Schutte.Correctness_E0

```
Instance Epsilon0_correct :
  ON_correct epsilon0 Epsilon0 (fun alpha => inject (cnf alpha)).
```

Project 4.2 *This exercise is a continuation of Project 3.11 on page 64.* Use `ON_mult` to define an ordinal notation `Omega2` for $\omega^2 = \omega \times \omega$.

Prove that `Omega2` is a sub-notation of `Epsilon0`.

Define on `Omega2` an addition compatible with the addition on `Epsilon0`.

Hint. You may use the following definition (in OrdinalNotations.Definitions).

```
Definition SubON_same_op  `{OA : @ON A ltA  compareA}
       `{OB : @ON B ltB  compareB}
       {iota : A -> B}
       {alpha: B}
       {_ : SubON OA OB alpha iota}
       (f : A -> A -> A)
       (g : B -> B -> B)
  :=
  forall x y,  iota (f x y) = g (iota x) (iota y).
```

Project 4.3 The class `ON` of ordinal notations has been defined long after this chapter, and is not used yet in the development of the type `E0`. A better integration of both notions should simplify the development on ordinals in Cantor normal form. This integration is planned for the future versions.

4.3 A Variant for Hydra Battles

4.3.1 Natural Sum (a.k.a. Hessenberg's Sum)

Natural sum (Hessenberg's sum) is a commutative and monotonous version of addition. It is used as an auxiliary operation for defining variants for hydra battles, where Hercules is allowed to chop off any head of the hydra.

In the litterature, the natural sum of ordinals α and β is often denoted by $\alpha\#\beta$ or $\alpha \oplus \beta$. Thus we called `oplus` the associated *Coq* function.

4.3.1.1 Definition of `oplus`

The definition of `oplus` is recursive in both of its arguments, which makes a structural recursive definition a little complex. We used the same pattern as for the `merge` function on lists of library `Coq.Sorting.Mergesort`.

1. Define a nested recursive function, using the `Fix` construct
2. Build a principle of induction dedicated to `oplus`
3. Establish equations associated to each case of the definition.

4.3.1.1.1 Nested recursive definition The following definition is composed of

- A main function `oplus`, structurally recursive in its first argument `alpha`
- An auxiliary function `oplus_aux` within the scope of `alpha`, structurally recursive in its argument `beta`; `oplus_aux beta` is supposed to compute `oplus alpha beta`.

From Module Epsilon0.Hessenberg

```
Fixpoint oplus (alpha beta : T1) : T1 :=
  let fix oplus_aux beta {struct beta} :=
      match alpha, beta with
        | zero, _ => beta
        | _,  zero => alpha
        | ocons a1 n1 b1, ocons a2 n2 b2 =>
          match compare a1 a2 with
            |  Gt => ocons a1 n1 (oplus b1 beta)
            |  Lt => ocons a2 n2 (oplus_aux b2)
            |  Eq => ocons a1 (S (n1 + n2)%nat) (oplus b1 b2)
          end
      end
  in oplus_aux beta.

Infix "o+" := oplus  (at level 50, left associativity).
```

The reader will note that each recursive call of the functions `oplus` and `oplus_aux` satisfies *Coq*'s constraint on recursive definitions. The function `oplus` is recursively called on a sub-term of its first argument, and `oplus_aux` on a sub-term of its unique argument. Thus, `oplus`'s definition is accepted by Coq as a structurally recursive function.

4.3.1.2 Rewriting Lemmas

Coq's constraints on recursive definitions result in the quite complex form of `oplus`'s definition. Proofs of properties of this function can be simpler if we *derive* rewriting lemmas that will help to simplify expressions of the form (`oplus` *a b*).

A first set of lemmas correspond to the various cases of `oplus`'s definition. They can be proved almost immediately, using `cbn` and `reflexivity` tactics.

```
Lemma oplus_alpha_0 (alpha : T1) : alpha o+ zero = alpha.
Proof.
  destruct alpha; reflexivity.
Qed.

Lemma oplus_0_beta (beta : T1): zero o+ beta = beta.
Proof.
  destruct beta; reflexivity.
Qed.
```

Project 4.4 Compare `oplus`'s definition (with inner fixpoint) with other possibilities (`coq-equations`, `Function`, etc.).

4.3.2 More Theorems on Hessenberg's Sum

We need to prove some properties of $\oplus$, particularly about its relation with the order $<$ on `T1`.

4.3.2.1 Boundedness

If α and β are both strictly less than ω^γ, then so is their natural sum $\alpha \oplus \beta$. This result can be proved by structural induction on γ.

```
Lemma lt_phi0_oplus : forall gamma alpha beta,
                        lt_phi0 alpha gamma ->
                        lt_phi0 beta gamma ->
                        lt_phi0 (alpha o+ beta) gamma.

Proof with auto.
  induction gamma; destruct alpha, beta.
(* ... *)
```

4.3.2.2 Commutativity, Associativity

We prove the commutativity of $\oplus$ in two steps.

First, we prove by transfinite induction on α that the restriction of $\oplus$ to the interval $[0..\alpha)$ is commutative.

```
Lemma oplus_comm_0 : forall alpha, nf alpha ->
     forall a b,  nf a -> nf b ->
                  lt a alpha ->
                  lt b alpha ->
                  a o+ b = b o+ a.
 Proof with eauto with T1.
    intros alpha Halph; transfinite_induction alpha.
(* rest of proof omitted *)
```

Then, we infer $\oplus$'s commutativity for any pair of ordinals: Let α and β be two ordinals strictly less than ϵ_0. Both ordinals α and β are strictly less than $\max(\alpha, \beta) + 1$. Thus, we have just to apply the lemma **oplus_comm_0**.

```
  Lemma oplus_comm : forall alpha beta,
      nf alpha -> nf beta ->
      alpha o+ beta =  beta o+ alpha.
  Proof with eauto with T1.
    intros alpha beta Halpha Hbeta;
    apply oplus_comm_0 with (succ (max alpha beta)) ...
  (* ... *)
```

The associativity of Hessenberg's sum is proved the same way.

```
 Lemma oplus_assoc_0 :
    forall alpha,
      nf alpha ->
      forall a b c,  nf a -> nf b -> nf c ->
                     lt a alpha ->
                     lt b alpha -> lt c alpha ->
                     a o+ (b o+ c) = (a o+ b) o+ c.
  Proof with eauto with T1.
    intros alpha Halpha.
```

```
    transfinite_induction alpha.
    (* ... *)
```

```
Lemma oplus_assoc : forall alpha beta gamma,
                      nf alpha -> nf beta -> nf gamma ->
                                  alpha o+ (beta o+ gamma) =
                                  alpha o+ beta o+ gamma.
Proof with eauto with T1.
   intros;
   apply oplus_assoc_0 with (succ (max alpha (max beta gamma))) ...
   (* ... *)
```

4.3.2.3 Monotonicity

At last, we prove that $\oplus$ is strictly monotonous in both of its arguments.

```
Lemma oplus_strict_mono_LT_l (alpha beta gamma : T1) :
  nf gamma   -> alpha  t1< beta ->
  alpha o+ gamma  t1< beta o+ gamma.

Lemma oplus_strict_mono_LT_r (alpha beta gamma : T1) :
  nf alpha -> beta t1< gamma ->
  alpha o+ beta t1< alpha o+ gamma.
```

Project 4.5 The library `Hessenberg` looks too long (proof scripts and compilation). Please try to make it simpler and more efficient! Thanks!

4.3.3 A Measure for Hydra-battle Termination

Let us define a measure from type `Hydra` into `T1`.

From Module Hydra.Hydra_Termination

```
Fixpoint m (h:Hydra) : T1 :=
  match h with head => zero
             | node hs => ms hs
end
with ms (s:Hydrae) :  T1 :=
  match s with  hnil => zero
              | hcons h s' => phi0 (m h) o+  ms s'
 end.
```

First, we prove that the measure $m(h)$ of any hydra h is a well-formed ordinal term of type `T1`.

```
Lemma m_nf : forall h, nf (m h).
Proof.
 intro h; elim h using Hydra_rect2
           with (P0 := fun s =>  nf (ms s)).
 (* ... *)
```

```
Lemma ms_nf : forall s, nf (ms s).
Proof with auto with T1.
(* ... *)
```

For proving the termination of all hydra battles, we have to prove that `m` is a variant. First, a few technical lemmas follow the decomposition of `round` into several relations. Then the lemma `round_decr` gathers all the cases.

```
Lemma S0_decr :
  forall s s', S0  s s' -> ms s' t1< ms s.
```

```
Lemma R1_decr : forall h h',
                  R1 h h' -> m h' t1< m h.
```

```
Lemma S1_decr n:
  forall s s', S1 n s s' -> ms s' t1<  ms s.
```

```
Lemma R2_decr n : forall h h', R2 n h h' -> m h'  t1< m h.
```

```
Lemma round_decr : forall h h', h -1-> h' -> m h' t1< m h.
Proof.
   destruct 1 as [n [H | H]].
   -  now apply R1_decr.
   -  now apply R2_decr with n.
Qed.
```

Finally, we prove the termination of all (free) battles.

```
Global Instance HVariant : Hvariant lt_wf free var.
Proof.
 split; intros; eapply round_decr; eauto.
Qed.

Theorem every_battle_terminates: Termination.
Proof.
  red; apply Inclusion.wf_incl with
         (R2 := fun h h' =>  m h t1< m h').
   red; intros;  now apply round_decr.
   apply Inverse_Image.wf_inverse_image, T1_wf.
Qed.
```

Conclusion

Let us recall three results we have proved so far.

- There exists a strictly decreasing variant which maps `Hydra` into the segment $[0, \epsilon_0)$ for proving the termination of any hydra battle
- There exists *no* such variant from `Hydra` into $[0, \omega^2)$, *a fortiori* into $[0, \omega)$.

So, a natural question is " Does there exist any strictly decreasing variant mapping type `Hydra` into some interval $[0, \alpha[$ (where $\alpha < \epsilon_0$) for proving the termination of all hydra battles". The next chapter is dedicated to a formal proof that there exists no such α, even if we consider a restriction to the set of "standard" battles.

Chapter 5

Strolling inside ϵ_0: The Ketonen-Solovay Machinery

5.1 Introduction

The reader may think that our proof of termination in the previous chapter requires a lot of mathematical tools and may be too complex. So, the question is "is there any simpler proof" ?

In their article [KP82], Kirby and Paris show that this result cannot be proved in Peano arithmetic. Their proof uses some knowledge about model theory and non-standard models of Peano arithmetic. In this chapter, we focus on a specific class of proofs of termination of hydra battles: construction of some variant mapping the type `Hydra` into a given initial segment of ordinals. Our proof relies only on the Calculus of Inductive Constructions and is a natural complement of the results proven in the previous chapter.

- There is no variant mapping the type `Hydra` into the interval $[0, \omega^2)$ (section 3.7.3 on page 55), and a fortiori $[0, \omega)$ (section 2.4.3 on page 39).
- There exists a variant which maps the type `Hydra` into the interval $[0, \epsilon_0)$ (theorem `every_battle_terminates`, in section 4.3.3 on page 86).

Thus, a very natural question is the following one:

> " Is there any variant from `Hydra` into some interval $[0, \mu)$, where $\mu < \epsilon_0$, for proving the termination of all hydra battles ?"

We prove in Coq the following result:

> There is no variant for proving the termination of all hydra battles from `Hydra` into the interval $[0..\mu)$, where $\mu < \epsilon_0$. The same impossibility holds even if we consider only standard battles (with the successive replication factors $0, 1, 2, \ldots, t, t+1, \ldots$).

Our proofs are constructive and require no axioms: they are closed terms of the CIC, and are mainly composed on function definitions and proofs of properties of these functions. They share much theoretical material with Kirby

and Paris', although they do not use any knowledge about Peano arithmetic nor model theory. The combinatorial arguments we use and implement come from an article by J. Ketonen and R. Solovay [KS81], already cited in the work by L. Kirby et J. Paris.Section 2 of this article: "A hierarchy of probably recursive functions", contains a systematic study of *canonical sequences*, which are closely related to rounds of hydra battles. Nevertheless, they have the same global structure as the simple proofs described in sections 2.4.3 on page 39 and 3.7.3 on page 55. We invite the reader to compare the three proofs step by step, lemma by lemma.

5.2 Canonical Sequences

Canonical sequences are functions that associate an ordinal $\{\alpha\}(i)$ to every ordinal $\alpha < \epsilon_0$ and positive integer i. They satisfy several nice properties:

- If $\alpha \neq 0$, then $\{\alpha\}(i) < \alpha$. Thus canonical sequences can be used for proofs by transfinite induction or function definition by transfinite recursion
- If λ is a limit ordinal, then λ is the least upper bound of the set $\{\{\lambda\}(i) \mid i \in \mathbb{N}_1\}$
- If $\beta < \alpha < \epsilon_0$, then there is a "path" from α to β, *i.e.* a sequence $\alpha_0 = \alpha, \alpha_1, \ldots, \alpha_n = \beta$, where for every $k < n$, there exists some i_k such that $\alpha_{k+1} = \{\alpha_k\}(i_k)$
- Canonical sequences correspond tightly to rounds of hydra battles: if $\alpha \neq 0$, then $\iota(\alpha)$ is transformed into $\iota(\{\alpha\}(i+1))$ in one round with the replication factor i (Lemma Hydra.O2H.canonS_iota_i).
- From the two previous properties, we infer that whenever $\beta < \alpha < \epsilon_0$, there exists a (free) battle from $\iota(\alpha)$ to $\iota(\beta)$.

Remark 5.1 In [KS81], canonical sequences are defined for any ordinal $\alpha < \epsilon_0$, by stating that if α is a successor ordinal $\beta + 1$, the sequence associated with α is simply the constant sequence whose terms are equal to β. Likewise, the canonical sequence of 0 maps any natural number to 0.

This convention allows us to make total the function that maps any ordinal α and natural number i to the ordinal $\{\alpha\}(i)$.

First, let us recall how canonical sequences are defined in [KS81]. For efficiency's sake, we decided not to implement directly K.&S's definitions, but to define in Gallina simply typed structurally recursive functions which share the abstract properties which are used in the mathematical proofs[1].

5.2.0.1 Mathematical Definition of Canonical Sequences

In [KS81] the definition of $\{\alpha\}(i)$ is based on the following remark:

> Any non-zero ordinal α can be decomposed in a unique way as the product $\omega^\beta \times (\gamma + 1)$.

[1] With a small difference: the 0-th term of the canonical sequence is not the same in our development as in [KS81].

Thus the $\{\alpha\}(i)$ s are defined in terms of this decomposition:

Definition 5.1 (Canonical sequences: mathematical definition)

- Let $\lambda < \epsilon_0$ be a limit ordinal
  - If $\lambda = \omega^{\alpha+1} \times (\beta + 1)$, then $\{\lambda\}(i) = \omega^{\alpha+1} \times \beta + \omega^{\alpha} \times i$
  - If $\lambda = \omega^{\gamma} \times (\beta+1)$, where $\gamma < \lambda$ is a limit ordinal, then $\{\lambda\}(i) = \omega^{\gamma} \times \beta + \omega^{\{\gamma\}(i)}$
- For successor ordinals, we have $\{\alpha + 1\}(i) = \alpha$
- Finally, $\{0\}(i) = \alpha$.

5.2.0.2 Canonical Sequences in Coq

Our definition may look more complex than the mathematical one, but uses plain structural recursion over the type **T1**. Thus, tactics like **cbn**, **simpl**, **compute**, etc., are applicable. For simplicity's sake, we use an auxiliary function `canonS` of type `T1 -> nat -> T1` such that (`canonS` α i) is equal to $\{\alpha\}(i + 1)$.

From Module Epsilon0.Canon

```
Fixpoint canonS alpha (i:nat) :=
  match alpha with
      zero => zero
    | ocons zero 0 zero => zero
    | ocons zero (S k) zero => FS k
    | ocons gamma 0 zero =>
      match pred gamma with
          Some gamma' => ocons gamma' i zero
        | None => ocons (canonS gamma i) 0 zero
      end
    | ocons gamma (S n) zero =>
       match pred gamma with
           Some gamma' => ocons gamma n (ocons gamma' i zero)
         | None => ocons gamma n (ocons (canonS gamma i) 0 zero)
       end
    | ocons alpha n beta => ocons alpha n (canonS beta i)
  end.
```

The following function computes $\{\alpha\}(i)$, except for the case $i = 0$, where it simply returns 0 [2].

```
Definition canon alpha i :=
  match i with 0 => zero | S j => canonS alpha j end.
```

[2]This restriction did not prevent us from proving all the main theorems of [KS81, KP82]. Nevertheless, in a future version of this development, we may define $\{\alpha\}(0)$ exactly as in [KS81]. But we are afraid this would be done at the cost of making some proofs much more complex.

For instance Coq's computing facilities allow us to verify the equalities $\{\omega^\omega\}(3) = \omega^3$ and $\{\omega^\omega * 3\}(42) = \omega^\omega * 2 + \omega^{42}$.

```
Compute (canon (omega ^ omega) 3).
```

```
  = phi0 (FS 2) : T1
```

```
Example canon3 :  canon (omega ^ omega) 3 = omega ^ 3.
Proof. reflexivity. Qed.
```

```
Compute pp (canon (omega ^ omega * 3) 42).
```

```
    = (omega ^ omega * 2 + omega ^ 42)%pT1
     : ppT1
```

Project 5.1 Many lemmas presented in this chapter were stated and proved before the introduction of the type class `ON` of ordinal notations, and in particular its instance `Epsilon0`. Thus definitions and lemmas refer to the type `T1` of possibly not well-formed terms. This should be fixed in a future version.

5.2.1 Basic Properties of Canonical Sequences

We did not try to prove that our definition really implements Ketonen and Solovay's [KS81]'s canonical sequences. The most important is that we were able to prove the abstract properties of canonical sequences that are really used in our proof. The complete proofs are in the module Epsilon0.Canon

Proving the equality $\{\alpha + 1\}(i) = \alpha$ is not as simple as suggested by the equations of definition 5.1. Nevertheless, we can prove it by plain structural induction on α.

```
Lemma canonS_succ i alpha :
  nf alpha ->  canonS (succ alpha) i = alpha.
Proof.
 induction alpha.
 (* ... *)
```

5.2.1.1 Canonical Sequences and the Order $<$

We prove by transfinite induction over α that $\{\alpha\}(i+1)$ is an ordinal strictly less than α (assuming $\alpha \neq 0$). This property allows us to use the function `canonS` and its derivates in function definitions by transfinite recursion.

```
Lemma canonS_LT i alpha :
  nf alpha -> alpha <> zero -> canonS alpha i t1<  alpha.
```

5.2.1.2 Limit Ordinals are Really Limits

The following theorem states that any limit ordinal $\lambda < \epsilon_0$ is the limit of the sequence $\{\lambda\}(i)\ (1 \leq i)$.

From Module Epsilon0.Canon

```
Lemma canonS_limit_strong (lambda : T1) :
     nf lambda ->
     limitb lambda  ->
     forall beta, beta t1< lambda ->
                  {i:nat | beta t1< canonS lambda i}.

Proof.
  transfinite_induction_LT lambda.
  (* ... *)
Defined.
```

Note the use of Coq's `sig` type in the theorem's statement, which relates the boolean function `limitb` defined on the `T1` data-type with a constructive view of the limit of a sequence: for any $\beta < \lambda$, we can compute an item of the canonical sequence of λ which is greater than β. We can also state directly that λ is a (strict) least upper bound of the elements of its canonical sequence.

```
Lemma canonS_limit_lub (lambda : T1) :
  nf lambda -> limitb lambda  ->
  strict_lub (fun i => canonS lambda i) lambda.
```

Exercise 5.1 Instead of using the `sig` type, define a simply typed function that, given two ordinals α and β, returns a natural number i such that, if α is a limit ordinal and $\beta < \alpha$, then $\beta < \{\alpha\}(i+1)$. Of course, you will have to prove the correctness of your function.

5.3 Accessibility inside ϵ_0 : Paths

Let us consider a kind of accessibility problem inside ϵ_0: given two ordinals α and β, where $\beta < \alpha < \epsilon_0$, find a *path* consisting of a finite sequence $\gamma_0 = \alpha, \ldots, \gamma_l = \beta$, where, for every $i < l$, $\gamma_i \neq 0$ [3] and there exists some strictly positive integer s_i such that $\gamma_{i+1} = \{\gamma\}(s_i)$,

Let s be the sequence $\langle s_0, s_1, \ldots, s_{l-1}\rangle$. We describe the existence of such a path with the notation $\alpha \underset{s}{\rightarrow} \beta$.

For instance, we have $\omega * 2 \xrightarrow[2,2,2,4,5]{} 3$, through the path $\langle \omega \times 2, \omega + 2, \omega + 1, \omega, 4, 3\rangle$.

Remark 5.2 Note that, given α and β, where $\beta < \alpha$, the sequence s which leads from α to β is not unique.

[3] This condition allows us to ignore paths which end by a lot of useless 0s.

Indeed, if α is a limit ordinal, the first element of s can be any integer i such that $\beta < \{\alpha\}(i)$, and if α is a successor ordinal, then the sequence s can start with any positive integer.

For instance, we have also $\omega * 2 \xrightarrow[3,4,5,6]{} \omega$. Likewise, $\omega * 2 \xrightarrow[1,2,1,4]{} 0$ and $\omega * 2 \xrightarrow[3,3,3,3,3,3,3]{} 0$.

5.3.1 Formal Definition

In Coq, the notion of path can be simply defined as an inductive predicate parameterized by the destination β.

From Module Epsilon0.Paths

```
(Definition transition_S i : relation T1 :=
  fun alpha beta =>  alpha <> zero /\ beta = canonS alpha i.

Definition transition i : relation T1 :=
  match i with 0 => fun _ _ => False | S j => transition_S j end.

Inductive path_to (beta: T1) : list nat -> T1 -> Prop :=
  path_to_1 : forall (i:nat) alpha ,
    i <> 0 ->
    transition i alpha beta ->
    path_to beta (i::nil) alpha
| path_to_cons : forall i alpha s gamma,
    i <> 0 ->
    transition i alpha gamma ->
    path_to beta  s gamma ->
    path_to beta  (i::s) alpha.
```

Remark 5.3 The definition above is parameterized with the *destination* of the path and indexed by the origin, hence the name `path_to`. The rationale behind this choice is a personal preference of the developer for the kind of eliminators generated by Coq in this case. The symmetric option could have been also considered (see also Remark 2.1 on page 27).

Remark 5.4 In the present version of our library, we use a variant `path_toS` of `path_to`, where the proposition (`path_toS` β s α) is equivalent to (`path_to` β `(shift` s`)` α). This variant is scheduled to be deprecated.

Exercise 5.2 Write a tactic for solving goals of the form (`path_to` β s α) where α, β and s are closed terms. You should solve automatically the following goals:

```
path_to omega (2::2::2::nil) (omega * 2).

path_to omega (3::4::5::6::nil) (omega * 2).

path_to zero (interval 3 14) (omega * 2).

path_to zero (repeat 3 8) (omega * 2).
```

5.3.2 Existence of a Path

By transfinite induction on α, we prove that for any $\beta < \alpha$, one can build a path from α to β (in other terms, β is accessible from α).

```
Lemma LT_path_to (alpha beta : T1) :
  beta t1< alpha -> {s : list nat | path_to beta s alpha}.
```

Exercise 5.3 (continuation of exercise 5.1 on page 93) Define a simply typed function for computing a path from α to β.

From the lemma `canonS_LT` 5.2.1.1 on page 92, we can convert any path into an inequality on ordinals (by induction on paths).

```
Lemma path_to_LT beta s alpha :
  path_to beta s alpha -> nf alpha -> beta t1< alpha.
```

5.3.3 Paths and Hydra Battles

In order to apply our knowledge about ordinal numbers less than ϵ_0 to the study of hydra battles, we define an injection from the interval $[0, \epsilon_0)$ into the type `Hydra`.

From Module Hydra.O2H

```
Fixpoint iota (alpha : T1) : Hydra :=
  match alpha with
  | T1.zero => head
  | ocons gamma n beta =>
        node (hcons_mult (iota gamma) (S n) (iotas beta))
  end
with iotas (alpha : T1) :  Hydrae :=
       match alpha with
       | T1.zero => hnil
       | ocons alpha0 n beta  =>
           hcons_mult (iota alpha0) (S n) (iotas beta)
       end.
```

For instance Fig. 5.1 shows the image by ι of the ordinal $\omega^{\omega+2}+\omega^{\omega}\times 2+\omega+1$

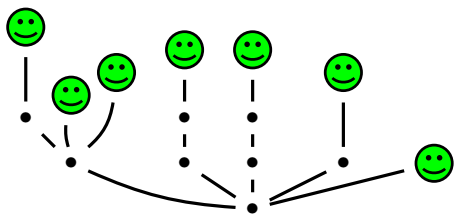

Figure 5.1: The hydra $\iota(\omega^{\omega+2} + \omega^{\omega} \times 2 + \omega + 1)$

The following lemma (proved in Hydra.O2H.v) maps canonical sequences to rounds of hydra battles.

```
Lemma canonS_iota i alpha :
    nf alpha -> alpha <> 0 ->
    iota alpha -1-> iota (canonS alpha i).
```

The next step of our development extends this relationship to the order $<$ on $[0, \epsilon_0)$ on one side, and hydra battles on the other side.

```
Lemma path_to_battle alpha s beta :
  path_to  beta  s alpha -> nf alpha ->
  iota alpha -+-> iota beta.
```

As a corollary, we are now able to transform any inequality $\beta < \alpha < \epsilon_0$ into a (free) battle.

```
Lemma LT_to_battle alpha beta :
    beta t1< alpha ->  iota alpha -+-> iota beta.
```

5.4 A Proof of Impossibility

We now have the tools for proving that there exists no variant bounded by some $\mu < \epsilon_0$ for proving the termination of all battles. The proof we are going to show is a proof by contradiction. It can be considered as a generalization of the proofs described in sections 2.4.3 on page 39 and 3.7.3 on page 55.

In the module Hydra.Epsilon0_Needed_Generic, we assume there exists some variant m bounded by some ordinal $\mu < \epsilon_0$. This part of the development is parameterized by some class B of battles, which will be instantiated later to `free` or `standard`.

```
Class BoundedVariant (B:Battle) :=
  {
    mu:T1 ;
    m: Hydra -> T1;
    mu_nf: nf mu;
    Hvar: Hvariant T1_wf B m;
    m_bounded: forall h, m h t1< mu
  }.
```

Let us assume there exists such a variant:

```
Section Bounded.
  Context (B: Battle) (Hy : BoundedVariant B).

  Hypothesis m_decrease : forall  i h h',
        round_n i h h' -> m h' t1< m h.
```

Remark 5.5 The hypothesis `m_decrease` is not provable in general, but is satisfied by the `free` and `standard` kinds of battles. This trick allows to "factorize" our proofs of impossibility.

First, we prove that $m(\iota(\alpha))$ is always greater than or equal to α, by transfinite induction over α.

```
Lemma m_ge_0 alpha:  nf alpha -> alpha t1<= m (iota alpha).
```

- If $\alpha = 0$, the inequality trivially holds
- If α is the successor of some ordinal β, the inequality $\beta \leq m(\iota(\beta))$ holds (by induction hypothesis). But the hydra $\iota(\alpha)$ is transformed in one round into $\iota(\beta)$, thus $m(\iota(\beta)) < m(\iota(\alpha))$. Hence $\beta < m(\iota(\alpha))$, which implies $\alpha \leq m(\iota(\alpha))$
- If α is a limit ordinal, then α is the least upper bound of the set of all the $\{\alpha\}(i)$. Thus, we have just to prove that $\{\alpha\}(i) < m(\iota(\alpha))$ for any i.
  - Let i be some natural number. By the induction hypothesis, we have $\{\alpha\}(i) \leq m(\iota(\{\alpha\}(i)))$. But the hydra $\iota(\alpha)$ is transformed into $\iota(\{\alpha\}(i))$ in one round, thus $m(\iota(\{\alpha\}(i))) < m(\iota(\alpha))$, by our hypothesis `m_decrease`.

Please note that the impossibility proofs of sections 2.4.3 on page 39 and 3.7.3 on page 55 contain a similar lemma, also called `m_ge`. We are now able to build a counter-example.

```
Definition big_h := iota mu.
Definition beta_h := m big_h.
Definition small_h := iota beta_h.
```

From Lemma `m_ge_0` we infer the following inequality :

```
Corollary m_ge_generic : m big_h t1<= m small_h.
```

The (big) rest of the proof is dedicated to prove formally the converse inequality `m small_h t1< m big_h`.

5.4.1 The case of Free Battles

Let us now consider that B is instantiated to `free` (which means that we are considering proofs of termination of *all* battles). The following lemmas are proved in Module Hydra.Epsilon0_Needed_Free. The case $B =$ `standard` is studied in section 5.5 on the following page.

```
Section Impossibility_Proof.

  Context (Var : BoundedVariant free ).
```

1. The following lemma is an application of `m_ge_generic`, since `free` satisfies trivially the hypothesis `m_decrease` (see page 96).

```
Lemma m_ge : m big_h t1<= m small_h.
  Proof.
    apply m_ge_generic.
   (* ... *)
```

2. From the hypothesis `m_bounded`, we have `m big_h t1< mu`

3. By Lemma `LT_to_battle`, we get a (free) battle from `big_h = iota mu` to `small_h = iota (m big_h)`.

```
  Lemma  big_to_small : big_h  -+-> small_h.
```

4. From the hypotheses on m, we infer:

```
Lemma m_lt : m small_h t1< m big_h.
```

5. From lemmas `m_ge` and `m_lt`, and the irreflexivity of $<$, we get a contradiction.

```
Theorem Impossibility_free : False.

End Impossibility_Proof.
```

We have now proved there exists no bounded variant for the class of free battles.

```
Check Impossibility_free.
```

```
  Impossibility_free
     : BoundedVariant free -> False
```

5.5 The Case of Standard Battles

One may wonder if our theorem holds also in the framework of standard battles. Unfortunately, its proof relies on the lemma `LT_to_round_plus` of Module Hydra.O2H.

```
Lemma LT_to_round_plus alpha beta :
    beta t1< alpha ->  iota alpha -+-> iota beta.
```

This lemma builds a battle out of any inequality $\beta < \alpha$. It is a straightforward application of `LT_path_to` of Module Epsilon0.Paths:

```
Lemma LT_path_to (alpha beta : T1) :
  beta t1< alpha -> {s : list nat | path_to beta s alpha}.
```

The sequence s, used to build the sequence of replication factors of the battle depends on β, so we cannot be sure that the generated battle is a genuine standard battle.

The solution of this issue comes once again from Ketonen and Solovay's article [KS81]. Instead of considering plain paths, i.e. sequences $\alpha_0 = \alpha, \alpha_1, \ldots, \alpha_k = \beta$ where α_{j+1} is equal to $\{\alpha_j\}(i_j)$ where i_j is *any* natural number, we consider various constraints on these sequences. In particular, a path is called *standard* if $i_{j+1} = i_j + 1$ for every $j < k$. It corresponds to a "segment" of some standard battles. Please note that the vocabulary on paths is ours, but all the concepts come really from [KS81].

In Coq, standard paths can be defined as follows.

From Module Epsilon0.KS

```
(**  standard path from (i, alpha) to (j, beta) *)

Inductive standard_pathR(j:nat)( beta:T1):  nat -> T1 -> Prop :=
  std_1 : forall i alpha,
       beta = canon alpha i -> j = S i ->
       standard_pathR j beta i  alpha
| std_S : forall i alpha,
      standard_pathR j beta (S i) (canon alpha i)  ->
      standard_pathR j beta i alpha.

Definition standard_path  i alpha j beta :=
   standard_pathR j beta i alpha.
```

In the mathematical text and figures, we shall use the notation $\alpha \underset{i,j}{\longrightarrow} \beta$ for the proposition (`standard_path` i α j β). In [KS81] the notation is $\alpha \xrightarrow[i]{*} \beta$ for the proposition $\exists j, i < j \wedge \alpha \underset{i,j}{\longrightarrow} \beta$.

Our goal is now to transform any inequality $\beta < \alpha < \epsilon_0$ into a standard path $\alpha \underset{i,j}{\longrightarrow} \beta$ for some i and j, then into a standard battle from $\iota(\alpha + i)$ to $\iota(\beta)$. Following [KS81], we proceed in two stages:

1. we simulate plain (free) paths from α to β with paths made of steps $(\gamma, \{\gamma\}(n))$, *with the same n all along the path*
2. we simulate any such path by a standard path.

5.5.1 Paths with a Constant Index

First of all, paths with a constant index enjoy nice properties. They are defined as paths where all the i_j are equal to the same natural number i, for some $i > 0$.

Like in [KS81], we shall use the notation $\alpha \xrightarrow[i]{} \beta$ for denoting such a path, also called an i-path.

```
Definition const_pathS i :=
    clos_trans_1n T1 (fun alpha beta => beta = canonS alpha i).

Definition const_path i alpha beta :=
```

```
  match i with
    0 => False
  | S j => const_pathS j alpha beta
end.
```

A most interesting property of i-paths is that we can "upgrade" their index, as stated by K.&S.'s Corollary 12.

```
Corollary Cor12 (alpha : T1) :  nf alpha ->
         forall beta i n, beta  t1< alpha  ->
                i < n ->
                 const_pathS i alpha beta ->
                 const_pathS n alpha beta.
Proof.
  transfinite_induction_lt alpha.
  (* (long) proof skipped *)
```

We also use a version of `Cor12` with large inequalities.

```
Corollary Cor12_1 (alpha : T1) :
  nf alpha ->
  forall beta i n, beta t1< alpha ->
                   i <= n ->
                   const_pathS i alpha beta ->
                   const_pathS n alpha beta.
```

5.5.1.1 Sketch of Proof of `Cor12`

We prove this lemma by transfinite induction on α. Let us consider a path $\alpha \underset{i}{\to} \beta$ $(i > 0)$. Its first step is the pair $(\alpha, \{\alpha\}(i))$, We have $\{\alpha\}(i) < \alpha$ and $\{\alpha\}(i) \underset{i}{\to} \beta$. Let n be any natural number such that $n > i$. By the induction hypothesis, there exists a path $\{\alpha\}(n) \underset{i}{\to} \beta$.

- If α is a successor ordinal $\gamma + 1$, then $\{\alpha\}(n) = \{\alpha\}(i) = \gamma$. Thus we have a path $\alpha \underset{n}{\to} \gamma \underset{n}{\to} \beta$
- If α is a limit ordinal, we apply the following theorem (numbered `2.4` in Ketonen and Solovay's article).

  ```
  Theorem Theorem_2_4 (lambda : T1) :
     nf lambda ->
     limitb lambda  ->
     forall i j, (i < j)%nat ->
                 const_pathS 0 (canonS lambda j)
                               (canonS lambda i).
  ```

  We build the following paths :

  1. $\alpha \underset{n}{\to} \{\alpha\}(n)$

2. $\{\alpha\}(n) \underset{1}{\rightarrow} \{\alpha\}(i)$ (by `Theorem_2_4`),
3. $\{\alpha\}(n) \underset{n}{\rightarrow} \{\alpha\}(i)$ (applying the induction hypothesis to the preceding path);
4. $\{\alpha\}(i) \underset{n}{\rightarrow} \beta$ (applying the induction hypothesis)
5. $\alpha \underset{n}{\rightarrow} \beta$ (by composition of 1, 3, and 4).

Remark 5.6 `Cor12` "casts" i-paths into n-paths for any $n > i$. But the obtained n-path can be much longer than the original i-path. The following exercise will give an idea of this increase.

Exercise 5.4 Prove that the length of the $i+1$-path from ω^ω to ω^i is $1 + (i + 1)^{(i+1)}$, for any i. Note that the i-path from ω^ω to ω^i is only one step long.

Why is `Cor12` so useful? Let us consider two ordinals $\beta < \alpha < \epsilon_0$. By induction on α, we decompose any inequality $\beta < \alpha$ into $\beta < \{\alpha\}(i) < \alpha$, where i is some integer. Applying collorary `Cor12'` we build a n-path from β to α, where n is the maximum of the indices i met in the induction.

Lemma 1, Section 2.6 of [KS81] is naturally expressed in terms of Coq's `sig` construct.

```
Lemma Lemma2_6_1 (alpha : T1) :
  nf alpha -> forall beta,  beta t1< alpha  ->
  {n:nat | const_pathS n alpha beta}.
Proof.
  transfinite_induction alpha.
  (* ... *)
```

Intuitively, lemma `L2_6_1` shows that if $\beta < \alpha < \epsilon_0$, then there exists a battle from $\iota(\alpha)$ to $\iota(\beta)$ where the replication factor is constant, although large enough.

5.5.2 Casting Paths with a Constant Index into a Standard Path

The article [KS81] contains the following lemma, the proof of which is quite complex, which allows to simulate i-paths by $[i+1, j]$-paths, where j is large enough.

```
(* Lemma 1 page 300 of [KS] *)

Lemma constant_to_standard_path
  (alpha beta : T1) (i : nat):
  nf alpha -> const_pathS i alpha beta -> zero  t1< alpha ->
  {l:nat | standard_path (S i) alpha j beta}.
```

5.5.2.1 Sketch of Proof of `constant_to_standard_path`

Our proof follows the proof by Ketonen and Solovay, including its organization as a sequence of lemma. Since it is a non-trivial proof, we will comment its main steps below.

Préliminaries

Please note that, given an ordinal α : `T1`, and two natural numbers i and l, there exists at most a standard path $\alpha \xrightarrow[i,i+l]{*} \beta$. The following function computes β from α, i and l.

```
Fixpoint standard_gnaw (i:nat)(alpha : T1)(l:nat):  T1  :=
  match l with
  | 0 => alpha
  | S m => standard_gnaw (S i) (canon alpha i) m
  end.
```

```
  Compute standard_gnaw 2 omega 15.
(*   = zero
     : T1 *)
Compute pp (standard_gnaw 2 (omega^omega)  10).
(*
= (omega + 7)%pT1
     : ppT1
*)
Compute pp (standard_gnaw 4 (omega^omega)  100).
(*
 = (omega ^ 3 * 4 + omega ^ 2 * 5 + omega * 3 + 39)%pT1
     : ppT1 *)
```

By transfinite induction over α, we prove that the ordinal 0 is reachable from any ordinal $\alpha < \epsilon_0$ by some standard path.

```
Lemma standard_path_to_zero :
  forall  alpha i, nf alpha ->
                  {j: nat | standard_path (S i) alpha j zero}.
```

Noq, let us consider two ordinals $\beta < \alpha < \epsilon_0$. Let p be some $(n+1)$-path from α to β.

```
 Section Constant_to_standard_Proof.

  Variables (alpha beta: T1) (n : nat).
  Hypotheses (Halpha: nf alpha) (Hpos : zero t1<  beta)
             (p : const_pathS n alpha  beta).
```

Applying `standard_path_to_zero`, 0 is reachable from α by some standard path (see figure 5.2 on the next page).

Since comparison on `T1` is decidable, one can compute the last step γ of the standard path from $(\alpha, n+1)$ such that $\beta \leq \gamma$. Let l be the length of the path from α to γ. This step of the proof is illustrated in figure 5.3 on the facing page.

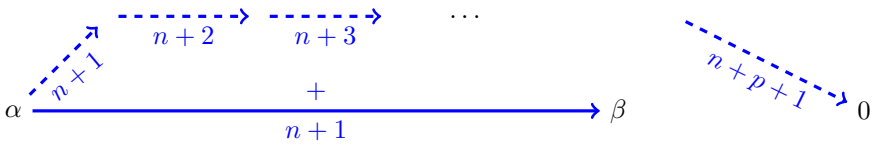

Figure 5.2: A nice proof (1)

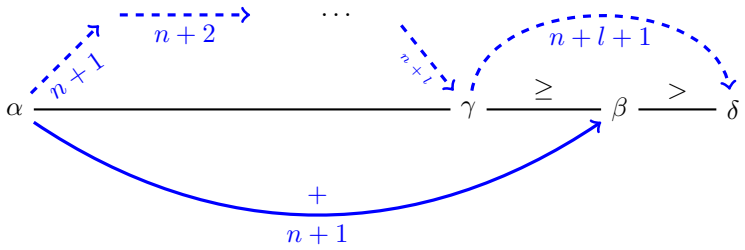

Figure 5.3: A nice proof (2)

- If $\beta = \gamma$, its OK! We have got a standard path from α to β with successive indices $n+1, n+2, \ldots, n+l+1$
- Otherwise, $\beta < \gamma$. Let us consider $\delta = \{\gamma\}(n+l+1)$. By applying several times lemma `Cor12`, one converts every path of Fig 5.3 into a $n+l+1$-path (see figure 5.4).

  But γ is on the $n+l+1$-path from α to β. As shown by figure 5.5 on the next page, the ordinal δ, reachable from γ in one single step, must be greater than or equal to β, which contradicts our hypothesis $\beta < \gamma$.

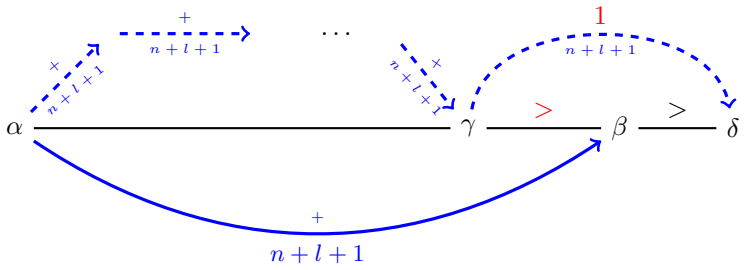

Figure 5.4: A nice proof (3)

The only possible case is thus $\beta = \gamma$, so we have got a standard path from α to β.

```
Lemma constant_to_standard_0 :
   {l : nat | standard_fun (S n) alpha l = beta}.
(* ... *)

End Constant_to_standard_Proof.
```

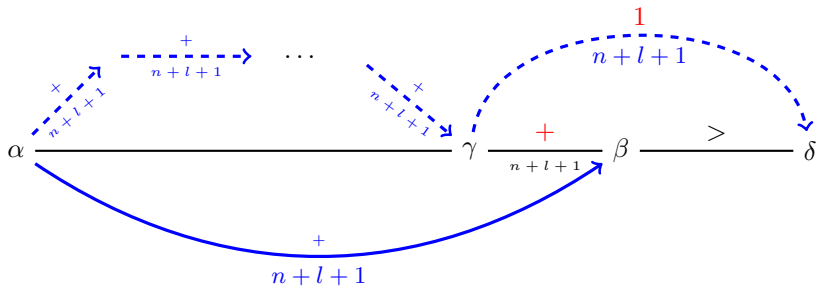

Figure 5.5: A nice proof (4)

Here is the full statement of the conversion from constant to standard paths.

```
Lemma constant_to_standard_path
  (alpha beta : T1) (i : nat):
  nf alpha -> const_pathS i alpha beta -> zero  t1< alpha ->
  {j:nat | standard_path (S i) alpha j beta}.
```

Applying `Lemma2_6_1` and `constant_to_standard_path`, we get the following corollary.

```
Corollary  LT_to_standard_path  (alpha beta : T1) :
  beta t1< alpha ->
  {n : nat & {j:nat | standard_path (S n) alpha j beta}}.
```

5.5.3 Back to Hydras

We are now able to complete our proof that there exists no bounded variant for proving the termination of standard hydra battles. This proof can be consulted in the module `../theories/html/hydras.Hydra.Epsilon0_Needed_Std.html`. Please note that it has the same global structure as in section5.4.1 Applying the lemmas `Lemma2_6_1` of the module `Lemma2_6_1` and `constant_to_standard_path`, we can convert any inequality $\beta < \alpha < \epsilon_0$ into a standard path from α to β, then into a fragment of a standard battle from $\iota(\alpha)$ to $\iota(\beta)$.

From Module Hydra.Epsilon0_Needed_Std

```
Lemma LT_to_standard_battle :
    forall alpha beta,
      beta t1< alpha ->
      exists n i,  battle standard  n (iota alpha) i (iota beta).
```

Next, please consider the following context:

```
Section Impossibility_Proof.

  Context (Var : BoundedVariant standard).
```

In the same way as for free battles, we import a large inequality from the module Hydra.Epsilon0_Needed_Generic.

```
Lemma m_ge : m big_h t1<= m small_h.
```

If remains to prove the following strict inequality, in order to have a contradiction.

```
Lemma m_lt : m small_h  t1< m big_h.
```

Sketch of proof: Let us recall that `big_h` $= \iota(\mu)$ and `small_h` $= \iota(m($`big_h`$))$.

Since $m($`big_h`$) < \mu$, there exists a standard path from μ to $m($`big_h`$)$, hence a standard battle from $\iota(\mu)$ to $\iota(m($`big_h`$))$, i.e. from `big_h` to `small_h`.

Since m is assumed to be a variant for standard battles, we get the inequality $m($`small_h`$) < m($`big_h`$)$.

5.5.4 Remarks

We are grateful to J. Ketonen and R. Solovay for the high quality of their explanations and proof details. Our proof follows tightly the sequence of lemmas in their article, with a focus on constructive aspects. Roughly steaking, our implementation *builds*, out of a hypothetic variant m, bounded by some ordinal $\mu < \epsilon_0$, a hydra `big_h` which verifies the impossible inequality $m($`big_h`$) < m($`big_h`$)$.

On may ask whether the preceding results are not too restrictive, since they refer to a particular data type `T1`. In fact, our representation of ordinals strictly less than ϵ_0 is faithful to their mathematical definition, at least Kurt Schütte's [Sch77], as proved in Chapter 7 on page 127. (please see also the module `Ordinals.Schutte.Correctness_E0`).

Thus, we can infer that our theorems can be applied to any well order.

Project 5.2 Study a possible modification of the definition of a variant (for standard battles).

- The variant is assumed to be strictly decreasing *on configurations reachable from some initial configuration where the replication factor is equal to* 0
- The variant may depend on the number of the current round.

In other words, its type should be `nat -> Hydra -> T1`, and it must verify the inequality $m\,(S\,i)\,h' < m\,i\,h$ whenever the configuration (i, h) is reachable from some initial configuration $(0, h_0)$ and `h` is transformed into `h'` in the considered round. Can we still prove the theorems of section 5.5 with this new definition?

Chapter 6

Large Sets and Rapidly Growing Functions

In this chapter, we try to feel how long a standard battle can be. To be precise, for any ordinal $\alpha < \epsilon_0$ and any positive integer k, we give a minoration of the number of steps of a standard battle which starts with the hydra $\iota(\alpha)$ and the replication factor k.

We express this number in terms of the Hardy hierarchy of fast-growing functions [BW85, Wai70, KS81, Prő13]. From the Coq user's point of view, such functions are very attractive: they are defined as functions in Gallina, and we can apply them *in theory*, but they are so complex that you will never be able to look at the result of the computation. Thus, our knowledge on these functions must rely on *proofs*. In our development, we use often the rewriting rules generated by Coq's `Equations` plug-in.

6.1 Definitions

Definition 6.1 *Let $0 < \alpha < \epsilon_0$ be any ordinal, and s a sequence of strictly positive natural numbers. We say that s is* minimally α-large *(in short: α-*mlarge*) if s if s is α-large and every strict prefix of s leads to a non-zero ordinal (*cf *Sect. 5.3.1 on page 94).*

From Module Epsilon0.Large_Sets

```
Definition mlarge alpha (s:list nat) := path_to zero s alpha.
```

Remark 6.1 Ketonen and Solovay [KS81] consider large finite *sets* of natural numbers, but they are mainly used as sequences. Thus, we chosed to represent them explicitely as (sorted) lists.

They also consider large (but not minimally large) sets. They can be defined in Coq as follows:
From Module Epsilon0.Paths

```
Fixpoint gnaw (alpha : T1) (s: list nat) :=
  match s with
```

```
    | nil => alpha
    | (0::s') => gnaw  alpha s'
    | (S i :: s')  =>  gnaw (canonS i alpha) s'
  end.

Definition large alpha (s:list nat) := gnaw alpha s = zero.
```

Let us consider two integers k and l, such that $0 < k < l$. In order to check whether the interval $[k, l]$ is minimally large for α, it is enough to follow from α the path associated with the interval $[k, l($ and verify that the last ordinal we obtain is equal to 1.

6.1.1 Example

For instance the interval $[6, 70]$ leads ω^2 to $\omega \times 2 + 56$. Thus this interval is not ω^2-mlarge.

```
Compute pp (gnaw (omega * omega) (interval 6 70)).
```

```
 = (omega * 2 + 56)%pT1
     : ppT1
```

Let us try another computation.

```
Compute (gnaw (omega * omega) (interval 6 700)).
```

```
 = zero : T1
```

We may say that the interval $[6, 700]$ is ω^2-large, since it leads to 0, but nothing assures us that the condition of minimality is satisfied.

The following lemma relates minimal largeness with the function `gnaw`.

```
Lemma mlarge_iff alpha x (s:list nat) :
  s <> nil -> ~ In 0 (x::s) ->
  mlarge alpha (x::s) <-> gnaw alpha (but_last x s) = one.
```

For instance, we can verify that the interval $[6, 510]$ is ω^2-mlarge.

From Module Epsilon0.Large_Sets_Examples

```
Example Ex1 : mlarge (omega * omega) (interval 6 510).
```

6.2 The Length of Minimal Large Sequences

Now, consider any natural number $k > 0$. We would like to compute a number l such that the interval $[k, l]$ is α-mlarge. So, the standard battle starting with $\iota(\alpha)$ and the replication factor k will end after $(l - k + 1)$ steps.

First, we notice that this number l exists, since the segment $[0, \epsilon_0)$ is well-founded and $\{\alpha\}(i) < \alpha$ for any i and $\alpha > 0$. Moreover, it is unique:

From Module Epsilon0.Large_Sets

```
Lemma mlarge_unicity alpha k l l' :
  mlarge alpha (interval (S k) l) ->
  mlarge alpha (interval (S k) l') ->
  l = l'.
```

Thus, it seems obvious that there must exist a function, parameterized by α which associates to any strictly positive integer k the number l such that the interval $[k, l]$ is α-mlarge. It would be fine to write in Gallina a definition like this:

```
Function L (alpha: E0) (i:nat) :  nat := ...
```

But we do not know how to fill the dots yet … In the next section, we will use Coq to reason about the *specification* of `L`, prove properties of any function which satisfies this specification. In Sect. 6.2.4, we use the `coq-equations` plug-in to define a function `L_`, and prove its correctness w.r.t. its specification.

Let $0 < \alpha < \epsilon_0$ be an ordinal term. We consider the functions which associate to each stritly positive integer k the number l, where the interval $[k, l)$ is α-mlarge.

Remark 6.2 The upper bound of the considered interval has been chosen to be $l - 1$ and not l, in order to simplify some statements and proofs in composition lemmas associated to ordinals of the form $\alpha \times i$ and $\omega^\alpha \times i + \beta$. In order to consider any ordinal below ϵ_0, we consider a special case for $\alpha = 0$.

6.2.1 Formal Specification

Our specification of the function `L` is as follows:

```
Inductive L_spec : T1 -> (nat -> nat) -> Prop :=
| L_spec0 : forall f, (forall k, f k = k) ->  L_spec zero f
| L_spec1 : forall alpha f,
    alpha <> zero ->
    (forall k, mlarge alpha (interval (S k) (Nat.pred (f (S k))))) ->
    L_spec alpha f.
```

Note that, for $\alpha \neq 0$, the value of $f(0)$ is not specified. Nevertheless, the restriction of f to the set of strictly positive integers is unique (up to extensionnality).

```
Lemma L_spec_unicity alpha f g :
  L_spec alpha f -> L_spec alpha g -> forall k, f (S k) = g (S k).
```

6.2.2 Abstract Properties

Let us now prove properties of any function f (if any) which satisfies `L_spec`. We are looking for properties which could be used for writing *equations* and prove the correctness of the function generated by the `coq-equations` plug-in. Moreover, they will give us some examples of L_α for small values of α.

Our exploration of the L_αs follows the usual scheme : transfinite induction, and proof-by-cases : zero, successors and limit ordinals.

6.2.2.1 The Ordinal Zero

The base case is directly a consequence of the specification.

```
Lemma L_zero_inv f : L_spec zero f -> forall k, f (S k) = S k.
```

6.2.2.2 Successor Ordinals

Let β be some ordinal, and assume the arithmetic function f satisfies the specification (`L_spec` β). Let k be any natural number. Any path from `succ` β to 0 starting at $k+1$ can be decomposed into a first step from `succ` β to β, then a path from β at $k+2$ to 0. By hypothesis the interval $[k+2, f(k+2)-1]$ is β-mlarge. But the interval $[k+1, f(k+2)-1]$ is the concatenation of the singleton $\{k+1\}$ and the interval $[k+2, f(k+2)-1]$. So, the function $\lambda\, k.\, f(k+1)$ satisfies the specification `L_spec` β.

Note that our decomposition of intervals works only if the intervals we consider are not empty. In order to ensure this property, we assume that $f\ k$ is always greater than k, which we note `S <<= f`, or `(fun_le S f)` (defined in Prelude.Iterates).

```
Definition fun_le f g  := forall n:nat,  f n <=  g n.
```

It looks also natural to show that the functions we consider are strictly monotonous. The section on successor ordinals has thus the following structure.

```
Section succ.
   Variables (beta : T1) (f : nat -> nat).

   Hypotheses (Hbeta : nf beta)
              (f_mono : strict_mono f)
              (f_Sle : S <<= f)
              (f_ok : L_spec beta f).

   Definition L_succ := fun k => f (S k).

   Lemma L_succ_mono : strict_mono L_succ.

   Lemma L_succ_Sle : S <<= L_succ.

   Lemma L_succ_ok : L_spec (succ beta) L_succ.

End succ.
```

6.2.2.3 Limit Ordinals

Let $\lambda < \epsilon_0$ be any limit ordinal. In a similar way as for successors, we decompose any path from λ (at k) into a step to $\{\lambda\}(k)$, then to 0. In the following section, we assume that there exists à correct function for $\{\lambda\}(k)$, *for any strictly positive* k.

```
Section lim.
  Variables (lambda : T1)
            (Hnf : nf lambda)
            (Hlim : limitb lambda)
            (f : nat -> nat -> nat)
            (H : forall k, L_spec (canonS lambda k) (f (S k))).

  Let L_lim k := f k (S k).

  Lemma L_lim_ok : L_spec lambda L_lim.

End lim.
```

6.2.3 First Results

Applying the previous lemmas on successors and limit ordinals, we get several correct implementations of `(L_spec` α`)` for small values of α.

6.2.3.1 Finite Ordinals

By iterating the functional `L_succ`, we get a realization of `(L_spec (fin` i`))` for any natural number i.

```
Definition L_fin i := fun k => (i + k)%nat.

Lemma L_fin_ok i : L_spec (fin i) (L_fin i).
```

6.2.3.2 The First Limit Ordinal ω

The lemmas `L_fin_ok` and `L_lim_ok` allow us to get by diagonalization a correct implementation for `L_spec omega`.

```
Definition L_omega k := S (2 * k)%nat.

Lemma L_omega_ok : L_spec omega L_omega.
```

6.2.3.3 Towards ω^2

We would like to get exact formulas for the ordinal ω^2, a.k.a. $\phi_0(2)$. This ordinal is the limit of the sequence $\omega \times i$ ($i \in \mathbb{N}$. Thus, we have to study ordinals of this form, then use our lemma on limits.

The following lemma establishes a path from $\omega \times (i+1)$ to $\omega \times i$.

```
Lemma path_to_omega_mult (i k:nat) :
  path_to (omega * i) (interval (S k) (2 * (S k))) (omega * (S i)).
```

Let us consider a path from $\omega \times (i+1)$ to 0 starting at $k+1$. A first "big step" will lead to $\omega \times i$ at $2(k+1)$. If $i > 0$, the next jump leads to $\omega \times (i-1)$ at $2(2(k+1))+1$, etc.

The following lemma expresses the length of the mlarge sequences associated with the finite multiples of ω.

```
Lemma omega_mult_mlarge_0 i  : forall k,
    mlarge  (omega * (S i))
            (interval (S k)
                      (Nat.pred (iterate (fun p =>  S (2 * p)%nat)
                                         (S i)
                                         (S k)))).
```

Thus, we infer the following result:
From Module Epsilon0.Large_Sets

```
Definition L_omega_mult i (x:nat) :=  iterate L_omega i x.

Lemma L_omega_mult_ok (i: nat) :  L_spec (omega * i) (L_omega_mult i).
```

For instance, let us consider the ordinal $\omega \times 8$, and a sequence starting at $k = 5$.

```
Compute L_omega_mult 8 5.
```

```
= 1535
     : nat
```

More generally, we prove the equality $L_{\omega \times i}(k) = 2^i \times (k+1) - 1$.

```
Lemma L_omega_mult_eqn (i : nat) :
  forall (k : nat),  (0 < k)%nat  ->
                     L_omega_mult i k = (exp2 i * S k - 1)%nat.
```

By diagonalization, we obtain a simple formula for L_{ω^2}.

```
Definition L_omega_square k := iterate (fun z => S (2 * z)%nat)
                                       k
                                       (S k).

Lemma L_omega_square_eqn k :
  (0 < k)%nat ->
  L_omega_square k = (exp2 k * (k + 2) - 1)%nat.


Lemma L_omega_square_ok: L_spec (omega * omega)
          L_omega_square.

Compute L_omega_square 8.
```

```
 = 2559
     : nat
```

6.2.3.4 Going Further

Let us consider a last example, "computing" L_{ω^3}. Since the canonical sequence associated with this ordinal is composed of the $\omega^2 \times i$ ($i \in \mathbb{N}_1$), we have to study this sequence.

To this end, we prove a generic lemma, which expresses $L_{\omega^\alpha \times i}$ as an iterate of L_{ω^α}. Note that in this lemma, we assume that the fonction associated with α is stritly monotonous and greater or equal than the successor function, and prove that $L_{\omega^\alpha \times i}$satisfies the same properties.

```
Section phi0_mult.
 Variables (alpha : T1) (f : nat -> nat).
 Hypotheses (Halpha : nf alpha)
            (f_mono : strict_mono f)
            (f_Sle : S <<= f)
            (f_ok : L_spec (phi0 alpha) f).

 Definition L_phi0_mult i := iterate f i.

Lemma L_phi0_mult_ok i:
  L_spec (ocons alpha i zero)  (L_phi0_mult (S i)).

 Lemma L_phi0_mult_smono i: strict_mono (L_phi0_mult i).

 Lemma L_phi0_mult_Sle i: S <<=  L_phi0_mult (S i).

End phi0_mult.
```

Let us look at the ordinal $\omega^2 \times i$, using `L_phi0_mult`

```
 Definition L_omega_square_times i :=  iterate L_omega_square i.

 Lemma L_omega_square_times_ok i :
    L_spec (ocons 2 i zero) (L_omega_square_times (S i)).
 Proof.
  apply L_phi0_mult_ok.
  -  auto with T1.
  -  apply L_omega_square_Sle.
  -  apply L_omega_square_ok.
 Qed.
```

We are now ready to get an exact formula for L_{ω^3}.

```
Definition L_omega_cube  := L_lim  L_omega_square_times .

Lemma L_omega_cube_ok : L_spec (phi0 3) L_omega_cube.
```

The function L_{ω^3} is just obtained by diagonalization upon $L_{\omega^2 \times i}$.

```
Lemma L_omega_cube_eqn i :
   L_omega_cube i = L_omega_square_times i (S i).
Proof. reflexivity. Qed.
```

Thus, for instance, $L_{\omega^3}(3) = L_{\omega^2\times 4}(3)$. Thus, we obtain an exact expression of this number.

```
Lemma L_omega_cube_3_eq:
   let N := exp2 95 in
   let P := (N * 97 - 1)%nat in
   L_omega_cube 3  =  (exp2 P * (P + 2) - 1)%nat.
```

This number is quite big. Using `Ocaml`'s `float` arithmetic, we can [under-]approximate it by $2^{3.8\times 10^{30}} \times 3.8 \times 10^{30}$.

```
# let exp2 x = 2.0 ** x;;

val exp2 : float -> float = <fun>
#   exp2 95.0 *. 97.0 -. 1.0;;
- : float = 3.84256588194182037e+30
# let n = exp2 95.0 ;;
# let p = n *. 97.0 -. 1.0;;
val p : float = 3.84256588194182037e+30

Estimation :
2 ** (3.84 e+30) * 3.84 e+30.
```

6.2.4 Using `Equations`

Note that we did not define any function L_α *for any* $\alpha < \epsilon_0$ yet. We have got no more than a collection of proved realizations of `L_spec` α for several values of α.

Using the `coq-equations` plug-in by M. Sozeau [SM19], we will now define a function `L_` which maps any ordinal $\alpha < \epsilon_0$ to a proven realization of `L_spec` α.

6.2.5 Definition

In order to get a total function, we use our type `E0` of well-formed ordinal terms,(see Sect 4.1.5.1 on page 73). Our definition is structured along a well-founded recursion and a case-study (zero, limit and successor ordinals).

From Module L_alpha).

```
From Equations Require Import Equations.
Require Import ArithRing Lia.

Instance Olt : WellFounded Lt := Lt_wf.
Hint Resolve Olt : E0.

(** Using Coq-Equations for building a function which satisfies
    Large_sets.L_spec *)

Equations  L_ (alpha: E0) (i:nat) :  nat  by wf  alpha Lt :=
  L_ alpha  i with E0_eq_dec alpha Zero :=
    { | left _ =>  i ;
      | right nonzero
```

```
          with Utils.dec (Limitb alpha) :=
          { | left _ =>  L_ (Canon alpha i)  (S i) ;
            | right notlimit =>  L_ (Pred alpha) (S i)}}.

Solve All Obligations with auto with E0.
```

It is worth looking at the answer from `Equations` and check (with `About`) all the lemmas this plug-in gives you for free. We show here only a part of Coq's anwer.

```
L__obligations_obligation_1 is defined
L__obligations_obligation_2 is defined
L__obligations is defined
L__clause_1 is defined
L__functional is defined
L_ is defined
...
L__equation_1 is defined
L__graph_mut is defined
L__graph_rect is recursively defined
L__graph_correct is defined
L__elim has type-checked, generating 1 obligation
L__elim is defined
FunctionalElimination_L_ is defined
FunctionalInduction_L_ is defined
```

Sometimes, these automatically generated statements may look cryptic.

```
About L__equation_1.
```

```
L__equation_1 :
forall (alpha : E0) (i : nat),
L_ alpha i = L__unfold_clause_1 alpha (E0_eq_dec alpha Zero) i
```

In most cases, it may be useful to write human-readable paraphrases of these statements.

```
Lemma L_zero_eqn : forall i, L_ Zero i = i.
Proof. intro i; now rewrite L__equation_1. Qed.

Lemma L_lim_eqn alpha i : Limitb alpha -> L_ alpha i =
                                          L_ (Canon alpha i) (S i).

Lemma L_succ_eqn alpha i :  L_ (Succ alpha) i = L_  alpha (S i).

Hint Rewrite L_zero_eqn L_succ_eqn : L_rw.
```

Using these three lemmas as rewrite rules, we can prove more properties of the functions `L_`α.

```
Lemma L_finite : forall i k :nat,  L_ i k = (i+k)%nat.
(* Proof by induction on i, using L_zero_eqn and L_succ_eqn *)

Lemma L_omega : forall k, L_ omega%e0 k = S (2 * k)%nat.
(* Proof using L_finite and L_lim_eqn *)
```

By well-founded induction on α, we prove the following lemmas:

```
Lemma L_ge_S alpha : alpha <> Zero -> S <<= L_ alpha.

Theorem L_correct alpha : L_spec (cnf alpha) (L_ alpha).
```

Please note that the proof of `L_correct` applies the lemmas proven in Sections 6.2.2.1, 6.2.2.2 and 6.2.2.3. Our previous study of `L_spec` allowed us to pave the way for the definition by `Equations` and the correctness proof.

6.2.5.1 Back to Hydra Battles

Theorem `battle_length_std` of Module Hydra.Hydra_Theorems relates the length of standard battles with the functions L_α.

```
Theorem battle_length_std (alpha : E0)  :
  alpha <> Zero ->
  forall k, (1 <= k)%nat ->
            battle_length standard k (iota (cnf alpha))
                          (L_ alpha (S k) - k).
```

Project 6.1 Instead of considering standard paths and battles, consider "constant" paths and the corresponding battles. Please use `Equations` in order to define the function that computes the length of the k-path which leads from α to 0. Prove a few exact formulas and minoration lemmas.

6.3 The Wainer-Hardy Hierarchy (Functions H_α)

In order to give an idea of the complexity of the functions L_αs, we compare them with a better known family of functions, the so called *Wainer-Hardy hierarchy* of fast growing functions, presented for instance in [Prő13].

For each ordinal α below ϵ_0, H_α is a total arithmetic function, defined by transfinite recursion on α, according to three cases:

- If $\alpha = 0$, then $H_\alpha(k) = k$ for any natural number k.
- If $\alpha = \mathrm{succ}(\beta)$, then $H_\alpha(k) = H_\beta(k+1)$ for any $k \in \mathbb{N}$
- If α is a limit ordinal, then $H_\alpha(k) = H_{(\{\alpha\}(k+1))}(k)$ for any $k \in \mathbb{N}$.

6.3.1 Hardy Functions in Coq

We define a function `H_` of type `E0 -> nat -> nat` by transfinite induction over the type `E0` of the well formed ordinals below ϵ_0.

From Module Epsilon0.H_alpha

```
Equations H_ (alpha: E0) (i:nat) :  nat  by wf  alpha lt :=
  H_ alpha  i with E0_eq_dec alpha Zero :=
    { | left _ =>  i ;
      | right nonzero
          with Utils.dec (Limitb alpha) :=
          { | left _ =>  H_ (Canon alpha (S i))  i ;
            | right notlimit =>  H_ (Pred alpha) (S i)}}.

Solve All Obligations with auto with E0.
```

```
Lemma H_eq1 : forall i, H_ Zero i = i.
Proof.   intro i; now rewrite H__equation_1.  Qed.

Lemma H_eq2 alpha i : Is_Succ alpha ->
                      H_ alpha i = H_ (Pred alpha) (S i).

Lemma H_eq3 alpha i : Limitb alpha ->
                      H_ alpha i =  H_ (Canon alpha (S i)) i.

Lemma H_eq4  alpha i :  H_ (Succ alpha) i = H_ alpha (S i).
```

6.3.2 First Steps of the Hardy hierarchy

Using rewrite rules from `H_eq1` to `H_eq4`, we can explore the functions H_α for some small values of α.

6.3.2.1 Finite Ordinals

By induction on i, we prove a simple expression of `H_ (Fin i)`, where `Fin` i is the i-th finite ordinal.

```
Lemma H_Fin : forall i k: nat,  H_ (Fin i) k = (i+k)%nat.
Proof with eauto with E0.
  induction i.
  - intros; simpl OF; simpl; autorewrite with H_rw E0_rw ...
  - intros ;simpl; autorewrite with H_rw E0_rw ...
    rewrite IHi; lia.
Qed.
```

6.3.2.2 Multiples of ω

Since the canonical sequence of ω is composed of finite ordinals, it is easy to get the formula associated with H_ω.

```
Lemma H_omega : forall k, H_ Omega k = S (2 * k)%nat.
Proof with auto with E0.
```

```
  intro k; rewrite H_eq3 ...
  - replace (Canon omega (S k)) with (Fin (S k)).
    + rewrite H_Fin; lia.
    +  now autorewrite with E0_rw.
Qed.
```

Before going further, we prove a useful rewriting lemma:

```
Lemma H_Plus_Fin alpha : forall i k : nat,
    H_ (alpha + i)%e0 k = H_ alpha (i + k)%nat.
(* Proof by induction on i *)
```

Then, we get easily formulas for $H_{\omega+i}$, and $H_{\omega\times i}$ for any natural number i.

```
Lemma H_omega_double k : H_ (omega * 2)%e0 k =  (4 * k + 3)%nat.
Proof.
 rewrite H_lim_eqn; simpl Canon.
 - ochange  (CanonS  (omega * 2)%e0 k)  (omega + (S k))%e0.
  + rewrite H_Plus_Fin, H_omega;  lia.
  -  now compute.
Qed.

Lemma H_omega_3 k : H_ (omega * 3)%e0 k = (8 * k + 7)%nat.

Lemma H_omega_4 k : H_ (omega * 4)%e0 k = (16 * k + 15)%nat.

Lemma H_omega_i i  : forall k,
    H_ (omega * i)%e0 k = (exp2 i * k + Nat.pred (exp2 i))%nat.
```

Crossing a new limit, we prove the following equality:

$$H_{\omega^2}(k) = 2^{k+1} \times (k+1) - 1$$

.

```
Lemma H_omega_sqr : forall k,
    H_ (Phi0  2)%e0 k = (exp2 (S k ) *  S k - 1)%nat.
Proof.
  intro k;
   rewrite H_lim_eqn; auto with E0.
  - ochange (Canon (Phi0 2) (S k)) (omega * (S k))%e0.
    +  rewrite H_omega_i; simpl (exp2 (S k)).
       *  rewrite Nat.add_pred_r.
          -- lia.
          --   generalize (exp2_not_zero k);  lia.
    + cbn; f_equal; lia.
Qed.
```

6.3.2.3 New Limits

Our next step would be to prove an exact formula for $H_{\omega^\omega}(k)$. Since the canonical sequence of ω^ω is composed of all the ω^i, we first need to express H_{ω^i} for any natural number i.

Let i and k be two natural numbers. The ordinal $\{\omega^(i+1)\}(k)$ is the product $\omega^i \times k$, so we need also to consider ordinals of this form.

1. First, we express $H_{\omega^\alpha \times (i+2)}$ in terms of $H_{\omega^\alpha \times (i+1)}$.

```
Lemma H_Omega_term_1 : alpha <> Zero -> forall  k,
    H_ (Omega_term alpha (S i)) k =
    H_ (Omega_term alpha i) (H_ (Phi0 alpha) k).
```

2. Then, we prove by induction on i that $H_{\omega^\alpha \times (i+1)}$ is just the $(i+1)$-th iterate of H_{ω^α}.

```
Lemma H_Omega_term (alpha : E0)  :
alpha <> Zero -> forall i k,
  H_ (Omega_term alpha i) k = iterate  (H_ (Phi0 alpha)) (S i) k.
```

3. In particular, we have got a formula for $H_{\omega^{i+1}}$.

```
Definition H_succ_fun f k := iterate f (S k) k.

Lemma H_Phi0_succ alpha  : alpha <> Zero -> forall k,
      H_ (Phi0 (Succ alpha)) k = H_succ_fun (H_ (Phi0 alpha)) k.

Lemma H_Phi0_Si : forall i k,
      H_ (Phi0 (S i)) k = iterate H_succ_fun i (H_ omega) k.
```

We get now a formula for H_{ω^3}:

```
Lemma H_omega_cube : forall k,
    H_ (Phi0 3)%e0 k = iterate (H_ (Phi0 2))  (S k) k.
Proof.
  intro k; rewrite <-FinS_eq, -> Fin_Succ, H_Phi0_succ; auto.
  compute; injection 1; discriminate.
Qed.
```

6.3.2.4 A Numerical Example

It seems hard to capture the complexity of this function by looking only at this "exact" formula. Let us consider a simple example, the number $H_{\omega^3}(3)$.

```
Section H_omega_cube_3.

Let f k :=   (exp2 (S k ) * (S k) - 1)%nat.

Remark R0 k :  H_ (Phi0 3)%e0 k = iterate f (S k) k.
```

Thus, the number $H_{\omega^3}(3)$ can be written as four nested applications of f.

```
Fact F0 : H_ (Phi0 3) 3 = f (f (f (f 3))).
 rewrite R0; reflexivity.
Qed.
```

In order to make this statement more readable, we can introduce a local définition.

```
Let N := (exp2 64 * 64 - 1)%nat.
```

This number looks quite big; let us compute an approximation in `Ocaml`:

```
# (2.0 ** 64.0 *. 64.0 -. 1.0);;
```

```
- : float = 1.18059162071741 13e+21
```

```
Fact F1 : H_ (Phi0 3) 3 = f (f N).
Proof.
 rewrite H_omega_cube_0; reflexivity.
Qed.


Lemma F1_simpl : H_ (Phi0 3) 3 =
                 (exp2 (exp2 (S N) * S N) * (exp2 (S N) * S N) - 1)%nat.
```

In a more classical writing, this number is displayed as follows:

$$H_{\omega^3}(3) = 2^{(2^{N+1}\times(N+1))} \times (2^{N+1} \times (N+1)) - 1$$

We leave as an exercise to determine the best approximation as possible of the size of this number (for instance its number of digits). For instance, if we do not take into account the multiplications in the formula above, we obtain that, in base 2, the number $H_{\omega^3}(3)$ has at least $2^{10^{21}}$ digits. But it is still an under-approximation !

```
End H_omega_cube_3.
```

Now, we have got at last an exact formula for H_{ω^ω}.

```
Lemma H_Phi0_omega : forall k, H_ (Phi0 omega) k =
                                iterate H_succ_fun  k (H_ omega) k.
Proof with auto with E0.
  intro k; rewrite H_lim_eqn, <- H_Phi0_Si ...
  -  rewrite CanonS_Canon, CanonS_Phi0_lim;  f_equal ...
Qed.
```

Using extensionality of the functional `iterate`, we can get a closed formula.

```
Lemma H_Phi0_omega_closed_formula k :
  H_ (Phi0 omega) k =
  iterate (fun (f: nat -> nat) (l : nat) => iterate  f (S l) l)
              k
              (fun k : nat => S (2 * k)%nat)
              k.
```

Note that this short formula contains two occurences of the functional `iterate`, the outer one is in fact a second-order iteration (on type `nat -> nat`) and the inner one first-order (on type `nat`).

6.3.3 Abstract Properties of H-functions

Since pure computation seems to be useless for dealing with expressions of the form $H_\alpha(k)$, even for small values of α and k, we need to prove theorems for comparing $H_\alpha(k)$ and $H_\beta(l)$, in terms of comparison between α and β on the one hand, k and l on the other hand.

But beware of non-theorems! For instance, one could believe that H is monotonous in its first argument. The following proof shows this is false.

```
Remark H_non_mono1 :
  ~ (forall alpha beta k, (alpha o<= beta)%e0 ->
                          (H_ alpha k <= H_ beta k)%nat).
Proof.
 intros H ;specialize (H 42 omega 3).
 assert (H0 :(42 o<= omega)%e0) by (repeat split; auto).
 apply H in H0; rewrite H_Fin, H_omega  in H0; lia.
Qed.
```

On the contrary, the fonctions of the Hardy hierarchy have the following five properties [KS81]: for any $\alpha < \epsilon_0$,

- the function H_α is strictly monotonous : For all $n, p \in \mathbb{N}, n < p \Rightarrow H_\alpha(n) < H_\alpha(p)$.
- If $\alpha \neq 0$, then for every n, $n < H_\alpha(n)$.
- The function H_α is pointwise less or equal than $H_{\alpha+1}$
- For any $n \geq 1$, $H_\alpha(n) < H_{\alpha+1}(n)$. *We say that* $H_{\alpha+1}$ *dominates* H_α *from* 1.
- For any n and β, if $\alpha \underset{n}{\rightarrow} \beta$, then $H_\beta(n) \leq H_\alpha(n)$.

In Coq, we follow the proof in [KS81]. This proof is mainly a single proof by transfinite induction on α of the conjonction of the five properties. For each α, the three cases : $\alpha = 0$, α is a limit, and α is a successor are considered. Inside each case, the five sub-properties are proved sequentially.

Definition 6.2 *Let f zand g be two arithmetic functions; f is said to* dominate *g if $f(p) > g(p)$ for any all sufficiently large p.*

```
Section Proof_of_Abstract_Properties.
  Record P (alpha:E0) : Prop :=
    mkP {
        PA : strict_mono (H_ alpha);
        PB : alpha <> Zero -> forall n,  (n < H_ alpha n)%nat;
        PC : H_ alpha <<= H_ (Succ alpha);
        PD : dominates_from 1 (H_ (Succ alpha)) (H_ alpha);
```

```
        PE : forall beta n, Canon_plus n alpha beta ->
                            (H_ beta n <= H_ alpha n)%nat}.


Theorem P_alpha : forall alpha, P alpha.
  Proof.
    intro alpha; apply well_founded_induction with lt.
   (* rest of proof skipped *)

Section Proof_of_Abstract_Properties.
```

6.3.4 Comparison between `L_` and `H_`

By well-founded induction on α, we prove that our L hierachy is "almost" the Hardy hierarchy (up to a small shift).

From Module Epsilon0.L_alpha

```
Theorem H_L_ alpha: forall i:nat,  (H_ alpha i <= L_ alpha (S i))%nat.
```

6.3.4.1 Back to Hydras

The following theorem relates the length of (standard) battles with the Hardy hierarchy.

From Module Epsilon0.L_alpha

```
Theorem battle_length_std_Hardy (alpha : E0) :
  alpha <> Zero ->
  forall k , 1 <= k -> exists l: nat,
       H_ alpha k - k <= l /\
       battle_length standard k (iota (cnf alpha)) l.
```

6.4 The Wainer Hierarchy (Functions F_α)

The Wainer hierarchy [BW85, Wai70, KS81], is also a family of fast growing functions, indexed by ordinals below ϵ_0, by the following equations:

- $F_0(i) = i + 1$
- $F_{\beta+1}(i) = (F_\beta)^{(i+1)}(i)$, where $f^{(i)}$ is the i-th iterate of f.
- $F_\alpha(i) = F_{\{\alpha\}(i)}(i)$ if α is a limit ordinal.

A first attempt is to write a definition of F_α by equations, in the same as for *H_alpha*. We use the functional `iterate` defined in Module Prelude.Iterates.

```
Fixpoint iterate {A:Type}(f : A -> A) (n: nat)(x:A) :=
  match n with
  | 0 => x
  | S p => f (iterate  f p x)
  end.
```

The following code comes from `../theories/html/hydras.Epsilon0.F_alpha.html`.

```
Fail Equations F_ (alpha: E0) (i:nat) :  nat  by wf  alpha Lt :=
  F_ alpha  i with E0_eq_dec alpha Zero :=
    { | left _ =>  i ;
      | right nonzero
          with Utils.dec (Limitb alpha) :=
          { | left _ =>  F_ (Canon alpha i)  i ;
            | right notlimit =>  iterate (F_ (Pred alpha))  (S i) i}}.
```

```
The command has indeed failed with message:
In environment
alpha : E0
notlimit : Limitb alpha = false
nonzero : alpha <> Zero
i : nat
F_ : forall x : E0, nat -> x o< alpha -> nat
The term "F_ (Pred alpha) ?x" has type "Pred alpha o< alpha -> nat"
while it is expected to have type "Pred alpha o< alpha -> Pred alpha o< alpha"
(cannot unify "nat" and "Pred alpha o< alpha").
```

We presume that this error comes from the recursive call of `F_` inside an application of `iterate`. The workaround we propose is to define first the iteration of `F_` as an helper F^*, then to define the function F as a "iterating F^* once".

`Equations` accepts the following definition, relying on lexicographic ordering on pairs (α, n).

```
Definition call_lt (c c' : E0 * nat) :=
  lexico Lt (Peano.lt) c c'.

Lemma call_lt_wf : well_founded call_lt.
  unfold call_lt; apply Inverse_Image.wf_inverse_image,  wf_lexico.
  -  apply E0.Lt_wf.
  -  unfold Peano.lt; apply Nat.lt_wf_0.
Qed.

Instance WF : WellFounded call_lt := call_lt_wf.

Equations  F_star (c: E0 * nat) (i:nat) :  nat by wf  c call_lt :=
    F_star (alpha, 0) i := i;
    F_star (alpha, 1) i
      with E0_eq_dec alpha Zero :=
           { | left _ => S i ;
             | right nonzero
                 with Utils.dec (Limitb alpha) :=
                 { | left _ => F_star (Canon alpha i,1) i ;
                   | right notlimit =>
                     F_star (Pred alpha, S i)  i}};
    F_star (alpha,(S (S n))) i :=
               F_star (alpha, 1) (F_star (alpha, (S n)) i).

(* Finally, F_ alpha is defined as its first iterate ! *)
```

```
Definition F_  alpha i := F_star (alpha, 1) i.
```

It is quite easy to prove that our function `F_` satisfies the equations on page 123.

```
Lemma F_zero_eqn : forall i, F_ Zero i = S i.

Lemma F_lim_eqn : forall alpha i,  Limitb alpha ->
                                F_ alpha i = F_ (Canon alpha i) i.

Lemma F_succ_eqn : forall alpha i,
    F_ (Succ alpha) i = iterate (F_ alpha) (S i) i.
```

As for the Hardy functions, we can use these equalities as rewrite rules for "computing" some values of $F_\alpha(i)$, for small values of α.

```
Lemma LF1 : forall n,  F_ 1 n = S (2 * n).

Lemma LF2 : forall i, (exp2 i * i < F_ 2 i)%nat.
```

Like in Sect 6.3.3, we prove by induction the following properties (see [KS81]).

```
Theorem F_alpha_mono alpha : strict_mono (F_ alpha).

Theorem F_alpha_ge_S alpha : forall n, (n < F_ alpha n)%nat.

Theorem F_alpha_Succ_le alpha : F_ alpha <<= F_ (Succ alpha).

Theorem F_alpha_dom alpha : dominates_from 1 (F_ (Succ alpha)) (F_ alpha).

Theorem F_alpha_beta alpha : forall beta n, Canon_plus n alpha beta ->
                                        (F_ beta n <= F_ alpha n)%nat.
```

As a corollary, we prove the following proposition, p. 284 of [KS81].

If $\beta < \alpha$, F_α dominates F_β.

```
Lemma Propp284 : forall alpha beta : E0,
   beta o< alpha -> dominates (F_ alpha) (F_ beta).
```

Exercise 6.1 Let us quote a theorem from [KS81] (page 297).

$$H_{\omega^\alpha}(n+1) \geq F_\alpha(n) \quad (n \geq 1, \alpha < \epsilon_0)$$
$$F_\alpha(n+1) \geq H_{\omega^\alpha}(n) \quad (n \geq 1, \alpha < \epsilon_0)$$

Thus H_{ω^α} and F_α have essentially the same order of growth.

But, before trying to prove these facts, look at the definition of function H in Ketonen and Solovay's paper ! Is it really the same as the definition we quote from Prőmel's chapter [Prő13], whereas [KS81] define $H_\alpha(n)$ as "the least integer k such that $[n, k]$ is α-large". Thus, it may be useful to adapt the statement above.

Exercise 6.2 Prove the following result [KS81](p. 298).

For $n \geq 2$ and $\alpha \geq 3$, $F_\alpha(n+1) \geq 2^{F_\alpha(n)}$.

Chapter 7

Kurt Schütte's Axiomatic Definition of Countable Ordinals

In the present chapter, we compare our implementation of the segment $[0, \epsilon_0)$ with a mathematical text in order to "validate" our constructions. Our reference here is the axiomatic definition of the set of countable ordinals, in chapter V of Kurt Schütte's book " Proof Theory " [Sch77].

Remark 7.1 *In all this chapter, the word "ordinal" will be considered as a synonymous of "countable ordinal"*

Schütte's definition of countable ordinals relies on the following three axioms: There exists a strictly ordered set , such that

1. $(\mathbb{O}, <)$ is well-ordered
2. Every bounded subset of $\mathbb{O}$ is countable
3. Every countable subset of $\mathbb{O}$ is bounded.

Starting with these three axioms, Schütte re-defines the vocabulary about ordinal numbers: the null ordinal 0, limits and successors, the addition of ordinals, the infinite ordinals ω, ϵ_0, Γ_0, etc.

This chapter describes an adaptation to Coq of Schütte's axiomatization. Unlike the rest of our libraries, our library Ordinals.Schutte is not constructive, and relies on several axioms.

- First, please keep in mind that the set of countable ordinals is not countable. Thus, we cannot hope to represent all countable ordinals as finite terms of an inductive type, which was possible with the set of ordinals strictly less than ϵ_0 (resp. Γ_0)
- We tried to be as close as possible to K. Schütte's text, which uses "classical" mathematics : excluded middle, Hilbert's ϵ (choice) and Russel's ι (definite description) operators. Both operators allow us to write definitions close to the natural mathematical language, such as "succ is *the* least ordinal strictly greater than α"

- Please note that only the library Schutte/*.v is "contaminated" by axioms, and that the rest of our libraries remain constructive.

7.1 Declarations and Axioms

Let us declare a type `Ord` for representing countable ordinals, and a binary relation `lt`. Note that, in our development, `Ord` is a type, while the *set* of countable ordinals (called $\mathbb{O}$ by Schütte) is the full set over the type `Ord`.

We use Florian Hatat's library on countable sets, written as he was a student of *École Normale Supérieure de Lyon*. A set A is countable if there is an injective function from A to $\mathbb{N}$ (see Library `Schutte.Countable`).

From Module `Schutte.Schutte_basics`

```
Parameter Ord : Type.
Parameter lt : relation Ord.
Infix "<" := lt : schutte_scope.

Definition ordinal := Full_set Ord.
```

Schütte's first axiom tells that `lt` is a well order on the set `ordinal` (The class `WO` is defined in Module `Well_Orders.v`).

```
Variables (M:Type)
         (Lt : relation M).

Class WO : Type:=
    {
      Lt_trans : Transitive  Lt;
      Lt_irreflexive : forall a:M, ~ (Lt a a);
      well_order : forall (X:Ensemble M)(a:M),
          In X a ->
          exists a0:M, least_member  X a0
    }.
```

```
  Axiom AX1 : WO lt.
```

The second and third axioms say that a subset X of $\mathbb{O}$ is (strictly) bounded if and only if it is countable.

```
Axiom AX2 : forall X: Ensemble Ord,
   (exists a,  (forall y, In X y -> y < a)) ->
   countable X.

Axiom AX3 : forall X : Ensemble Ord,
              countable X ->
              exists a,  forall y, In X y -> y < a.
```

`AX2` and `AX3` could have been replaced by a single axiom (using the `iff` connector), but we decide to respect as most as possible the structure of Schütte's definitions.

7.2 Additional Axioms

The adaptation of Schütte's mathematical discourse to Coq led us to import a few axioms from the standard library. We encourage the reader to consult Coq's FAQ about the safe use of axioms `https://github.com/coq/coq/wiki/The-Logic-of-Coq#axioms`.

7.2.0.1 Classical Logic

In order to work with classical logic, we import the module Coq.Logic.Classical of Coq's standard library, specifically the following axiom:

```
Axiom classic : forall P:Prop, P \/ ~P.
```

7.2.0.2 Description Operators

In order to respect Schütte's style, we imported also the library `Coq.Logic.Epsilon`. The rest of this section presents a few examples of how Hilbert's choice operator and Church's definite description allow us to write understandable definitions (close to the mathematical natural language).

7.2.0.3 The Definition of zero

According to the definition of a well order, every non-empty subset of `Ord` has a least element. Furthermore, this least element is unique.

```
Remark R : exists! z : Ord, least_member lt  ordinal z.
Proof.
  destruct inh_Ord as [a]; apply (well_order (WO:=AX1)) with a .
  split.
Qed.
```

Assume we want to call this element `zero`.

```
Definition zero : Ord.
Proof.
  Fail destruct R.
```

```
The command has indeed failed with message:
Case analysis on sort Type is not allowed for inductive
definition ex.
```

Indeed, the basic logic of Coq does not allow us to eliminate a proof of a proposition $\exists\, x : A,\ P(x)$ for building a term whose type lies in the sort `Type`. The reasons for this impossibility are explained in many documents [BC04, Chl11, Coq].

Let us import the library `Coq.Logic.Epsilon`, which contains the following axiom and lemmas.

```
Axiom epsilon_statement:
  forall (A : Type) (P : A->Prop), inhabited A ->
    {x : A | (exists x, P x) -> P x}.
```

Hilbert's ϵ *operator* is derived from this axiom.

```
  Definition epsilon (A : Type) (i:inhabited A) (P : A->Prop) : A
  := proj1_sig (epsilon_statement P i).

Lemma constructive_indefinite_description :
  forall (A : Type) (P : A->Prop),
    (exists x, P x) -> { x : A | P x }.
```

If we consider the *unique existential* quantifier $\exists!$, we obtain Church's *definite description operator*.

```
Definition iota (A : Type) (i:inhabited A) (P : A->Prop) : A
  := proj1_sig (iota_statement P i).
```

```
 Lemma constructive_definite_description :
  forall (A : Type) (P : A->Prop),
    (exists! x, P x) -> { x : A | P x }.
```

```
Definition iota_spec (A : Type) (i:inhabited A) (P : A->Prop) :
  (exists! x:A, P x) -> P (iota i P)
  := proj2_sig (iota_statement P i).
```

Indeed, the operators `epsilon` and `iota` allowed us to make our definitions quite close to Schütte's text. Our libraries `Schutte.MoreEpsilonIota` and `Schutte.PartialFun` are extensions of `Coq.logic.Epsilon` for making easier such definitions. See also an article in french [Cas07].

```
Class InH (A: Type) : Prop :=
   InHWit : inhabited A.

Definition some {A:Type} {H : InH A} (P: A -> Prop) :=
   epsilon (@InHWit A H) P.

Definition the {A:Type} {H : InH A} (P: A -> Prop) :=
   iota (@InHWit A H) P.
```

In order to use these tools, we had to tell Coq that the type `Ord` is not empty:

```
Axiom inh_Ord : inhabited Ord.
```

We are now able to define `zero` as the least ordinal. For this purpose, we define a function returning the least element of any [non-empty] subset.

```
Definition the_least {M: Type} {Lt}
          {inh : InH M} {WO: WO Lt} (X: Ensemble M)  : M :=
  the  (least_member  Lt X ).
```

From Module `Schutte.Schutte_basics`

```
Definition zero: Ord :=the_least ordinal.
```

We want to prove now that zero is less than or equal to any ordinal number.

```
Lemma zero_le (alpha : Ord) :  zero <= alpha.
Proof.
  unfold zero, the_least, the; apply iota_ind.
```

According to the use of the description operator `iota`, we have to solve two trivial sub-goals.

1. Prove that there exists a unique least member of `Ord`
2. Prove that being a least member of `Ord` entails the announced inequality

```
2 subgoals (ID 155)

  alpha : Ord
  ============================
  exists ! x : Ord, least_member lt ordinal x

subgoal 2 (ID 156) is:
 forall a : Ord, unique (least_member lt ordinal) a ->
                 a <= alpha
```

```
  - apply the_least_unicity, Inh_ord.
  - destruct 1 as [[_ H1] _]; apply H1; split.
Qed.
```

7.2.0.4 Remarks on `epsilon` and `iota`

What would happen in case of a misuse of `epsilon` or `iota` ? For instance, one could give a unsatisfiable specification to `epsilon` or a specification for `iota` that admits several realizations.

Let us consider an example:

```
Module Bad.

 Definition bottom := the_least (Empty_set Ord).
```

```
 bottom is defined
```

Since we won't be able to prove the proposition `{exists! a: Ord, least_member (Empty_set Ord) a}`, the only properties we would be able to prove about `bottom` would be *trivial* properties, *i.e.*, satisfied by *any* element of type `Ord`, like for instance `bottom = bottom`, or `zero <= bottom`.

```
Lemma le_zero_bottom : zero <= bottom.
Proof. apply zero_le. Qed.

Lemma bottom_eq : bottom = bottom.
Proof. trivial. Qed.

Lemma le_bottom_zero : bottom <= zero.
Proof.
   unfold bottom, the_least, the; apply iota_ind.
```

```
2 subgoals (ID 413)

  ============================
  exists ! x : Ord, least_member lt (Empty_set Ord) x

subgoal 2 (ID 414) is:
 forall a : Ord, unique (least_member lt (Empty_set Ord)) a ->
      a <= zero
```

```
Abort.
End Bad.
```

In short, using `epsilon` and `iota` in our implementation of countable ordinals after Schütte has two main advantages.

- It allows us to give a *name* (using `Definition`) two witnesses of existential quantifiers (let us recall that, in classical logic, one may consider non-constructive proofs of existential statements)
- By separating definitions from proofs of [unique] existence, one may make definitions more concise and readable. Look for instance at the definitions of `zero`, `succ`, `plus`, etc. in the rest of this chapter.

7.3 The Successor Function

The definition of the function `succ:Ord -> Ord` is very concise. The successor of any ordinal α is the smallest ordinal strictly greater than α.

```
Definition succ (alpha : Ord) := the_least (fun beta => alpha < beta).
```

Using `succ`, we define the folloing predicates.

```
Definition is_succ (alpha:Ord) := exists beta, alpha = succ beta.

Definition is_limit (alpha:Ord) := alpha <> zero /\ ~ is_succ alpha.
```

How do we prove properties of the successor function? First, we make its specification explicit.

```
Definition succ_spec (alpha:Ord) :=
  least_member   lt (fun z => alpha < z).
```

Then, we prove that our function succ meets this specification.

```
Lemma succ_ok : forall alpha,  succ_spec alpha  (succ alpha).
Proof.
  intros; unfold succ, the_least, the;  apply iota_spec.
```

```
1 subgoal (ID 172)

  alpha : Ord
  ============================
  exists ! x : Ord, succ_spec alpha x
```

We have now to prove that the set of all ordinals strictly greater than α has a unique least element. But the singleton set $\{\alpha\}$ is countable, hence bounded (by the axiom AX3). Hence; the set $\{\beta \in \mathbb{O} | \alpha < \beta\}$ is not empty and therefore has a unique least element.

The Coq proof script is quite short.

```
  destruct (@AX3 (Singleton _ alpha)).
  - apply countable_singleton.
  -  unfold succ_spec; apply the_least_unicity;  exists x; intuition.
Qed.
```

We can "uncap" the description operator for proving properties of the succ function.

```
Lemma lt_succ (alpha : Ord) :  alpha < succ alpha.
Proof.
  destruct  (succ_ok  alpha);  tauto.
Qed.

Hint Resolve lt_succ : schutte.

Lemma lt_succ_le (alpha beta : Ord):
  alpha < beta -> succ alpha <= beta.
Proof with eauto with schutte.
  intros  H;  pattern (succ alpha); apply the_least_ok ...
  exists (succ alpha); red;apply lt_succ ...
Qed.
```

```
Lemma lt_succ_le_2 (alpha beta : Ord):
  alpha < succ beta -> alpha <= beta.

Lemma succ_mono (alpha beta : Ord):
  alpha < beta -> succ alpha < succ beta.

Lemma succ_monoR (alpha beta : Ord) :
```

```
 succ alpha < succ beta -> alpha < beta.

Lemma lt_succ_lt (alpha beta : Ord) :
  is_limit beta ->  alpha < beta -> succ alpha < beta.
```

7.4 Finite Ordinals

Using `succ`, it is now easy to define recursively all the finite ordinals.

```
Reserved Notation "'F' n" (at level 29) .

Fixpoint finite (i:nat) : Ord :=
  match i with
            | 0 => zero
            | S i => succ (F i)
  end
where "'F' i" := (finite i)  : schutte_scope.

Coercion finite : nat >-> Ord.
```

7.5 The Definition of `omega`

In order to define ω, the first infinite ordinal, we use an operator which "returns" the least upper bound (if it exists) of a subset $X \subseteq \mathbb{O}$. For that purpose, we first use a predicate: (`is_lub` D lt X a) if a belongs to D and is the least upper bound of X (with respect to lt).

```
Definition is_lub (M:Type)
                  (D : Ensemble M)
                  (lt : relation M)
                  (X:Ensemble M)
                  (a:M) :=
   In _ D a  /\ upper_bound  D lt X a  /\
   (forall y, In _ D y -> upper_bound  D lt X y  ->
                  y = a \/ lt a y).
```

```
Definition sup_spec X lambda := is_lub ordinal lt X lambda.

Definition sup (X: Ensemble Ord) : Ord  := the  (sup_spec X).

Notation "'|_|' X" := (sup X) (at level 29) : schutte_scope.
```

Then, we define the function `omega_limit` which returns the least upper bound of the (denumerable) range of any sequence `s: nat -> Ord`. By `AX3` this range is bounded, hence the set of its upper bounds is not empty and has a least element.

```
Definition omega_limit (s:nat->Ord) : Ord
  := |_| (seq_range s).
```

Then we define omega as the limit of the sequence of finite ordinals.

```
Definition _omega := omega_limit finite.

Notation "'omega'" := (_omega) : schutte_scope.
```

Among the numerous properties of the ordinal ω, les us quote the following ones (proved in Module Schutte.Schutte_basics)

```
Lemma finite_lt_omega : forall i: nat,  i < omega.

Lemma lt_omega_finite alpha : Ord) :
  alpha < omega ->  exists i:nat, alpha =  i.

Lemma is_limit_omega : is_limit omega.
```

7.5.1 Ordering Functions and Ordinal Addition

After having defined the finite ordinals and the infinite ordinal ω, we define the sum $\alpha+\beta$ of two countable ordinals. Schütte's definition looks like the following one:

> "$\alpha + \beta$ is the β-th ordinal greater than or equal to α"

The purpose of this section is to give a meaning to the construction "the α-th element of X" where X is any non-empty subset of $\mathbb{O}$. We follow Schütte's approach, by defining the notion of *ordering functions*, a way to associate a unique ordinal to each element of X. Complete definitions and proofs can be found in Module Schutte.Ordering_Functions).

7.5.2 Definitions

A *segment* is a set A of ordinals such that, whenever $\alpha \in A$ and $\beta < \alpha$, then $\beta \in A$; a segment is *proper* if it strictly included in $\mathbb{O}$.

```
 Definition segment (A: Ensemble Ord) :=
  forall alpha beta, In A alpha -> beta < alpha -> In A  beta.

Definition proper_segment (A: Ensemble Ord) :=
  segment A /\ ~ Same_set A ordinal.
```

Let A be a segment, and B a subset of $\mathbb{O}$: an *ordering function for A and B* is a strictly increasing bijection from A to B. The set B is said to be an *ordering segment* of A. Our definition in Coq is a direct translation of the mathematical text of [Sch77].

```
Definition ordering_function (f : Ord -> Ord)(A B : Ensemble Ord) :=
 segment A /\
 (forall a, In A a -> In B (f a)) /\
 (forall b, B b -> exists a, In A a /\ f a = b) /\
 forall a b, In A a -> In A b -> a < b ->  f a < f b.
```

```
Definition ordering_segment (A B : Ensemble Ord) :=
  exists f : Ord -> Ord, ordering_function f A B.
```

We are now able to associate with any subset B of $\mathbb{O}$ its ordering segment and ordering function.

```
Definition the_ordering_segment (B : Ensemble Ord) :=
  the  (fun x => ordering_segment x B).

Definition ord  (B : Ensemble Ord) :=
  some (fun f => ordering_function f (the_ordering_segment B) B).
```

Thus (`ord` $B\ \alpha$) is the α-th element of B. Please note that the last definition uses the epsilon-based operator `some` and not `the`. This is due to the fact that we cannot prove the unicity (w.r.t. Leibniz' equality) of the ordering function of a given set. By contrast, we admit the axiom `Extensionality_Ensembles`, from the library Coq.Sets.Ensembles, so we use the operator `the` in the definition of `the_ordering_segment`.

One of the main theorems of `Ordering_Functions` associates a unique segment and a unique (up to extensionality) ordering function to every subset B of $\mathbb{O}$.

```
About ordering_function_ex.
```

```
forall B : Ensemble Ord,
 exists ! S : Ensemble Ord,
      exists f : Ord -> Ord, ordering_function f S B
```

```
ordering_function_unicity :
forall (B S1 S2 : Ensemble Ord) (f1 f2 : Ord -> Ord),
ordering_function f1 S B ->
ordering_function f2 S2 B ->
fun_equiv f1 f2 S1 S2
```

Thus, our function `ord` which enumerates the elements of B is defined in a non-ambiguous way. Let us quote the following theorems (see Library `Schutte.Ordering_Functions` for more details).

```
Theorem ordering_le : forall f A B,
    ordering_function f A B ->
    forall alpha, In A alpha -> alpha <= f alpha.

Th_13_5_2 :
forall (A B : Ensemble Ord) (f : Ord -> Ord),
ordering_function f A B -> Closed B -> continuous f A B
```

7.5.3 Ordinal Addition

We are now ready to define and study addition on the type `Ord`. The following definitions and proofs can be consulted in Module `Schutte.Addition.v`.

```
Definition plus alpha := ord  (ge alpha).
Notation "alpha + beta " := (plus alpha beta) : schutte_scope.
```

In other words, $\alpha + \beta$ is the β-th ordinal greater than or equal to α. Thanks to generic properties of ordering functions, we can show the following properties of addition on $\mathbb{O}$. First, we prove a useful lemma:

```
Lemma plus_elim (alpha : Ord) :
  forall P : (Ord->Ord)->Prop,
    (forall f: Ord->Ord,
        ordering_function f ordinal (ge alpha)-> P f) ->
    P (plus alpha).
```

```
Lemma alpha_plus_zero (alpha: Ord): alpha + zero = alpha.
Proof.
 pattern  (plus alpha); apply plus_elim; eauto.
```

```
1 subgoal (ID 24)

  alpha : Ord
  ============================
  forall f : Ord -> Ord,
  ordering_function f ordinal (ge alpha) ->
  f zero = alpha
```

```
 (* rest of proof skipped *)
```

The following lemmas are proved the same way.

```

Lemma zero_plus_alpha (alpha : Ord) : zero + alpha = alpha.

Lemma le_plus_l (alpha beta : Ord) : alpha <= alpha + beta.

Lemma le_plus_r (alpha beta : Ord) :  beta <= alpha + beta.

Lemma plus_mono_r (alpha beta gamma : Ord) :
    beta < gamma -> alpha + beta < alpha + gamma.

Lemma plus_of_succ (alpha beta : Ord) :
    alpha + (succ beta) = succ (alpha + beta).

Theorem plus_assoc (alpha beta gamma : Ord) :
  alpha + (beta + gamma) = (alpha + beta) + gamma.

Lemma one_plus_omega :  1 + omega = omega.
```

```
Lemma finite_plus_infinite (n : nat) (alpha : Ord) :
  omega <= alpha -> n + alpha = alpha.
```

It isinteresting to compare the proof of these lemmas with the computational proofs of the corresponding statements in Module `Epsilon0.T1`. For instance, the proof of the lemma `one_plus_omega` uses the continuity of ordering functions (applied to `(plus 1)`) and compares the limit of the ω-sequences $i_{(i\in\mathbb{N})}$ and $(1+i)i_{(i\in\mathbb{N})}$, whereas in the library `Epsilon0/T1`, the equality $1+\omega=\omega$ is just proved with `reflexivity`!

7.5.3.1 Multiplication by a Natural Number

The multiplication of an ordinal by a natural number is defined in terms of addition. This operation is useful for the study of Cantor normal forms.

```
Fixpoint mult_Sn (alpha:Ord)(n:nat){struct n}: Ord :=
 match n with
            | 0 => alpha
            | S p => mult_Sn  alpha p + alpha
 end.

Definition mult_fin_r alpha n :=
  match n with
      0 => zero
    | S p => mult_Sn alpha p
  end.

Notation "alpha * n" := (mult_fin_r alpha n) : schutte_scope.
```

7.6 The Exponential of Basis ω

In this section, we define the function which maps any $\alpha \in \mathbb{O}$ to the ordinal ω^α, also written $\phi_0(\alpha)$. It is an opportunity to apply the definitions and results of the preceding section. Indeed, Schütte first defines a subset of $\mathbb{O}$: the set of additive principal ordinals, and ϕ_0 is just defined as the ordering function of this set.

7.6.1 Additive Principal Ordinals

Definition 7.1 *A non-zero ordinal α is said to be* additive principal *if, for all $\beta<\alpha$, $\beta+\alpha$ is equal to α. We call `AP` the set of additive principal ordinals.*

From Module `Schutte.AP`

```
Definition AP : Ensemble Ord :=
  fun alpha =>
  zero < alpha /\
  (forall beta, beta < alpha ->  beta + alpha = alpha).
```

7.6.2 The Function phi0

Let us call ϕ_0 the ordering function of AP. In the mathematical text, we shall use indifferently the notations ω^α and$\phi_0(\alpha)$.

```
Definition phi0 := ord AP.

Notation "'omega^'" := phi0 (only parsing) : schutte_scope.
```

7.6.3 Omega-towers and the Ordinal ϵ_0

Using ϕ_0, we can define recursively the set of finite omega-towers.

```
Fixpoint omega_tower (i : nat) : Ord :=
  match i with
    0 =>  1
  | S j => phi0 (omega_tower j)
  end.
```

Then, the ordinal ϵ_0 is defined as the limit of the sequence of all finite towers (a kind of infinite tower).

```
Definition epsilon0 := omega_limit omega_tower.
```

The rest of our library AP is devoted to the proof of properties of additive principal ordinals, hence of the ordering function $\phi 0$ and the ordinal ϵ_0 (which we could not express within the type T1).

7.6.4 Properties of the Set AP

The set of additive principal ordinals is not empty: it contains at least the ordinals 1 and ω.

```
Lemma AP_one : In AP 1.

Lemma AP_omega : In AP omega.
```

Moreover, 1 is the least principal ordinal and ω is the second element of AP.

```
Lemma least_AP: least_member  lt AP 1.

Lemma omega_second_AP :
  least_member   lt
                  (fun alpha => 1 < alpha /\ In AP alpha)
                  omega.
```

The set AP is *closed* under addition, and unbounded.

```
Lemma AP_plus_closed (alpha beta gamma : Ord):
     In AP alpha -> beta < alpha -> gamma < alpha ->
     beta + gamma < alpha.

Theorem AP_unbounded : Unbounded AP.
```

Finally, AP is (topologically) *closed* and ordered by the segment of all countable ordinals.

```
Definition Closed (B : Ensemble Ord) : Prop :=
  forall M, Included M B -> Inhabited _ M ->
                countable M -> In B (|_| M).
```

```
Theorem AP_closed : Closed AP.

Lemma AP_o_segment :  the_ordering_segment AP = ordinal.
```

7.6.4.1 Properties of the Function ϕ_0

The ordering function ϕ_0 of the set AP is defined on the full set $\mathbb{O}$ and is continuous (Schütte calls such a function *normal*).

```
Theorem normal_phi0 : normal phi0 AP.
```

The following properties come from the definition of ϕ_0 as the ordering function of AP. It may be interesting to compare these proofs with the computational ones described in Chapter 4.

```
Lemma AP_phi0 (alpha : Ord) : In AP (phi0 alpha).

Lemma phi0_zero : phi0 zero =  1.

Lemma phi0_mono (alpha beta : Ord) :
  alpha < beta ->  phi0 alpha < phi0 beta.

Lemma phi0_inj (alpha beta : Ord) :
    phi0 alpha = phi0 beta -> alpha = beta.

Lemma phi0_sup : forall (U: Ensemble Ord),
   Inhabited _ U ->   countable U ->  phi0 (|_| U) = |_| (image U phi0).

Lemma is_limit_phi0 (alpha : Ord) :
  zero < alpha ->  is_limit (phi0 alpha).

Lemma omega_eq : omega = phi0 1.

Lemma phi0_le (alpha : Ord) : alpha <= phi0 alpha.
```

Please note that the lemma omega_eq above, is consistent with the interpretation of the ordering function ϕ_0 as the exponential of basis ω. Indeed we could have written this lemma with our alternative notation:

```
 Lemma omega_eq : omega = omega^ 1.
```

7.7 More about ϵ_0

Let us recall that the limit ordinal ϵ_0 cannot be written within the type `T1`. Since we are now considering the set of all countable ordinals, we can now prove some properties of this ordinal.

We prove the inequality $\alpha < \omega^\alpha$ whenever $\alpha < \epsilon_0$. *Note that this condition was implicit in Module* ***`Epsilon0.T1`****.*

```
Lemma lt_phi0 (alpha : Ord):
  alpha < epsilon0 -> alpha < phi0 alpha.
```

The proof is as follows:

1. Since $\alpha < \epsilon_0$, consider the least i such that α is strictly less than the omega-tower of height i.
2. - If $i = 0$, then the result is trivial (because $\alpha = 0$)
   - Otherwise let $i = j+1$; α is greater than or equal to the omega-tower of height j. By monotonicity, $\phi_0(\alpha)$ is greater than or equal to the omega-tower of height $j + 1$, thus strictly greater than α

Moreover, ϵ_0 is the least ordinal α that verifies the equality $\alpha = \omega^\alpha$, in other words the least fixpoint of the function ϕ_0.

```
Theorem epsilon0_lfp : least_fixpoint lt phi0 epsilon0.
```

7.8 Critical Ordinals

For any (countable) ordinal α, the set $Cr(\alpha)$ is inductively defined as follows by Schütte (p.81 of [Sch77]).

- $Cr(0)$ is the set AP of additive principal ordinals.
- If $0 < \alpha$, then $Cr(\alpha)$ is the intersection of all the sets of fixpoints of the $Cr(\beta)$ for $\beta < \alpha$.

This definition is translated in Coq in Module `Schutte.Critical`, as the least fixpoint of a functional.

```
Definition Cr_fun : forall alpha : Ord,
       (forall beta : Ord, beta < alpha -> Ensemble Ord) ->
        Ensemble Ord
:=
   fun (alpha :Ord)
        (Cr : forall beta,
                beta < alpha -> Ensemble Ord)
        (x : Ord) => (
       (alpha = zero /\ AP x) \/
       (zero < alpha /\
        forall beta (H:beta < alpha),
          the_ordering_segment (Cr beta H) x /\ ord (Cr  beta H) x = x)).

Definition Cr (alpha : Ord) : Ensemble Ord :=
    (Fix  all_ord_acc (fun (_:Ord) => Ensemble Ord) Cr_fun) alpha.
```

```
Definition phi (alpha : Ord) : Ord -> Ord
    :=  ord (Cr alpha).

Definition A (alpha : Ord) : Ensemble Ord :=
  the_ordering_segment (Cr alpha).
```

For instance, we prove that $Cr(0)$ is the set of additive principals and that ϵ_0 belongs to $Cr(1)$.

```
Lemma Cr_zero_AP :  Cr 0 = AP

Lemma epsilon0_Cr1 : In (Cr 1) epsilon0.
```

Exercise 7.1 Prove that ϵ_0 is the least element of $Cr(1)$.

7.8.1 A flavor of Infinity

The family of the $Cr(\alpha)$s is made of infinitely many unbounded (hence infinite) sets. Let us quote Lemma 5, p. 82 of [Sch77]:

> For all α, the set $Cr(\alpha)$ is closed (for the least upper bound of non-empty countable sets) and unbounded.

We prove this result by transfinite induction on α of both properties. The proof is still quite long, by transfinite induction over α.

```
Section Proof_of_Lemma5.
  Let P (alpha:Ord) := Unbounded (Cr alpha) /\ Closed (Cr alpha).

 Lemma Lemma5 : forall alpha, P alpha.
(* ... *)
 End Proof_of_Lemma5.

Corollary Unbounded_Cr alpha : Unbounded (Cr alpha).
Proof.
  now destruct (Lemma5 alpha).
Qed.

Corollary Closed_Cr alpha : Closed (Cr alpha).
Proof.
  now destruct (Lemma5 alpha).
Qed.
```

7.9 Cantor Normal Form

The notion of Cantor normal form is defined for all countable ordinals. Nevertheless, note that, contrary to the implementation based on type `T1`, the Cantor normal form of an ordinal α may contain α as a sub-term[1].

[1]This would prevent us from trying to represent Cantor normal forms as finite trees (like in Sect. 4.1.2)

We represent Cantor normal forms as lists of ordinals. A list l is a Cantor normal form of a given ordinal α if it satisfies two conditions:

- The list l is sorted (in decreasing order) w.r.t. the order $\leq$
- The sum of all the ω^{β_i} where the β_i are the terms of l (in this order) is equal to α.

From `Schutte.CNF`

```
 Definition cnf_t := list Ord.

Fixpoint eval (l : cnf_t) : Ord :=
  match l with nil => zero
               | beta :: l' => phi0 beta + eval l'
  end.

Definition sorted (l: cnf_t) :=
  LocallySorted (fun alpha beta => beta <= alpha) l.

Definition is_cnf_of (alpha : Ord)(l : cnf_t) : Prop :=
  sorted l /\ alpha = eval l.
```

By transfinite induction on α, we prove that every countable ordinal α has at least a Cantor normal form.

```
Theorem cnf_exists (alpha : Ord) :
  exists l: cnf_t, is_cnf_of alpha l.
```

By structural induction on lists, we prove that this normal form is unique.

```
 Lemma cnf_unicity : forall l alpha,
   is_cnf_of alpha l ->
   forall l',  is_cnf_of alpha l' -> l=l'.
Proof.
 induction l.
 (*  ...  *)

Theorem cnf_exists_unique (alpha:Ord) :
  exists! l: cnf_t, is_cnf_of alpha l.
```

Finally, the following two lemmas relate ϵ_0 with Cantor normal forms.

If $\alpha < \epsilon_0$, then the Cantor normal form of α is made of ordinals strictly less than α.

```
Lemma cnf_lt_epsilon0 :
 forall l alpha,
   is_cnf_of alpha l ->  alpha < epsilon0 ->
   Forall (fun beta =>  beta < alpha) l.
```

Exercise 7.2 Please consider the following statement :

```
Lemma cnf_lt_epsilon0_iff :
 forall l alpha,
   is_cnf_of alpha l ->
   (alpha < epsilon0 <->  Forall (fun beta =>  beta < alpha) l).
```

Is it true ?

Finally, the Cantor normal form of ϵ_0 is just ω^{ϵ_0}.

```
Lemma cnf_of_epsilon0 : is_cnf_of epsilon0 (epsilon0 :: nil).
Proof.
  split.
  - constructor.
  - simpl;  now rewrite alpha_plus_zero, epsilon0_fxp.
Qed.
```

Project 7.1 Implement pages 82 to 85 of [Sch77] (critical, strongly critical, maximal critical ordinals, Feferman's ordinal Γ_0).

Remark 7.2 The sub-directory `theories/Gamma0` contains an (incomplete, still undocumented) implementation of the set of ordinals below Γ_0, represented in Veblen normal form.

7.10 An Embedding of `T1` into `Ord`

Our library `Schutte.Correctness_E0` establishes the link between two very different modelizations of ordinal numbers. In other words, it "validates" a data structure in terms of a classical mathematical discourse considered as a model. First, we define a function from `T1` into `Ord` by structural recursion.

```
Fixpoint inject (t:T1) : Ord :=
 match t with
     | T1.zero => zero
     | T1.ocons a n b =>  AP.phi0 (inject a) * S n + inject b
 end.
```

This function enjoys good commutation properties with respect to the main operations which allow us to build Cantor normal form.

```
Theorem inject_of_zero : inject T1.zero = zero.

Theorem inject_of_finite (n : nat):
  inject (T1.fin n) =  n.

Theorem inject_of_phi0 (alpha : T1):
  inject (phi0 alpha) = AP.phi0 (inject alpha).

Theorem inject_plus (alpha beta : T1): nf alpha -> nf beta ->
  inject (alpha + beta)%t1 = inject alpha + inject beta.
```

```
Theorem inject_mult_fin_r (alpha : T1)  :
  nf alpha -> forall n:nat , inject (alpha *  n)%t1 =  inject alpha * n.

Theorem inject_mono (beta gamma : T1) :
  T1.lt  beta gamma ->
  T1.nf beta -> T1.nf gamma ->
  inject beta < inject gamma.

Theorem inject_injective (beta gamma : T1) : nf beta -> nf gamma ->
  inject beta = inject gamma -> beta = gamma.
```

Finally, we prove that `inject` is a bijection from the set of all terms of `T1` in normal form to the set `members epsilon0` of the elements of `Ord` strictly less than ϵ_0.

```
Theorem inject_lt_epsilon0 (alpha : T1):
      inject alpha < epsilon0.

Theorem embedding :
     fun_bijection (nf: Ensemble T1)  (members epsilon0) inject.
```

7.10.1 Remarks

Let us recall that the library `Schutte` depends on five *axioms* and lies explicitly in the framework of classical logic with a weak version of the axiom of choice (please look at the documentation of `Coq.Logic.ChoiceFacts`). Nevertheless, the other modules: `Epsilon0`, `Hydra`, et `Gamma0` do not import any axioms and are really constructive.

Project 7.2 There is no construction of ordinal multiplication in [Sch77]. It would be interesting to derive this operation from Schütte's axioms, and prove its consistence with multiplication in ordinal notations for ϵ_0 and Γ_0.

7.11 Related Work

In [Gri13], José Grimm establishes the consistency between our ordinal notations (`T1` and `T2` (Veblen normal form) and his implementation of ordinal numbers after Bourbaki's set theory.

Chapter 8

The Ordinal Γ_0 (first draft)

This chapter and the files it presents are still very incomplete, considering the impressive properties of Γ_0 *[Gal91]. We hope to add new material soon, and accept contributions!*

8.1 Introduction

We present a notation system for the ordinal Γ_0, following Chapter V, Section 14 of [Sch77]: "A notation system for the ordinals $< \Gamma_0$". We try to be as close as possible to Schütte's text and usual practices of Coq developments.

The ordinal Γ_0 is defined in Section 13 of [Sch77] as the least *strongly critical ordinal*. It is widely known as the *Feferman-Schütte ordinal*.

Section V, 13 of [Sch77] defines *strongly critical* and *maximal α-critical* ordinals:

- α is strongly critical if α is α-critical,
- γ is maximal α-critical if γ is α-critical, and, for all $\xi > \alpha$, γ is not ξ-critical.

From `Schutte.Critical`

```
Definition strongly_critical alpha := In (Cr alpha) alpha.

Definition maximal_critical alpha : Ensemble Ord :=
  fun gamma =>
    In (Cr alpha) gamma /\
    forall xi, alpha < xi -> ~ In (Cr xi) gamma.

Definition Gamma0 := the_least strongly_critical.
```

Project 8.1 Prove that a (countable) ordinal α is strongly critical iff $\phi_\alpha(0) = \alpha$ (Theorem 13.13 of [Sch77]).

Project 8.2 Prove that the set of strongly critical ordinals is unbounded and closed (Theorem 13.14 of [Sch77]). Thus this set is not empty, hence has a least element. Otherwise, the definition of Γ_0 above would be useless.

In the present version of this development, we only study Γ_0 as a notation system, much more powerful than the ordinal notation for ϵ_0.

8.2 The Type T2 of Ordinal Terms

The notation system for ordinals less than γ_0 comes from the following theorem of [Sch77], where $\psi\,\alpha$ is the ordering function of the set of maximal α-critical ordinals.

> Any ordinal $\neq 0$ which is not strongly critical can be expressed in terms of $+$ and ψ.

Project 8.3 This theorem is not formally proved in this development yet. It should be!

Like in Chapter 4, we define an inductive type with two constructors, one for 0, the other for the construction $\psi(\alpha,\beta)\times(n+1)+\gamma$, adapting a Manolios-Vroon-like notation [MV05] to *Veblen normal forms*.
From `Gamma0.T2`

```
(**  [gcons alpha beta n gamma] is : [psi(alpha,beta)*(S n)+ gamma]  *)

Inductive T2 : Set :=
| zero : T2
| gcons : T2 -> T2  -> nat -> T2 -> T2.

Notation "[ alpha , beta ]" := (gcons alpha beta 0 zero)
                                  (at level 0): t2_scope.
```

Like in chapter 4, we get familiar with the type T2 by recognising simple constructs like finite ordinals, ω, etc., as inhabitants of T2.

```
Notation  "'one'"  := [zero,zero] : T2_scope.

(** The (n+1)-th finite ordinal *)
Notation "'FS' n" := (gcons zero zero n zero) (at level 10) : T2_scope.

(** the [n]-th ordinal  *)
Definition fin (n:nat) := match n with 0 => zero | S p => FS p end.

Notation "'omega'"  := [zero,one] : T2_scope.
```

```
Notation "'epsilon0'"  := ([one,zero]) : T2_scope.

Definition epsilon alpha := [one, alpha].
```

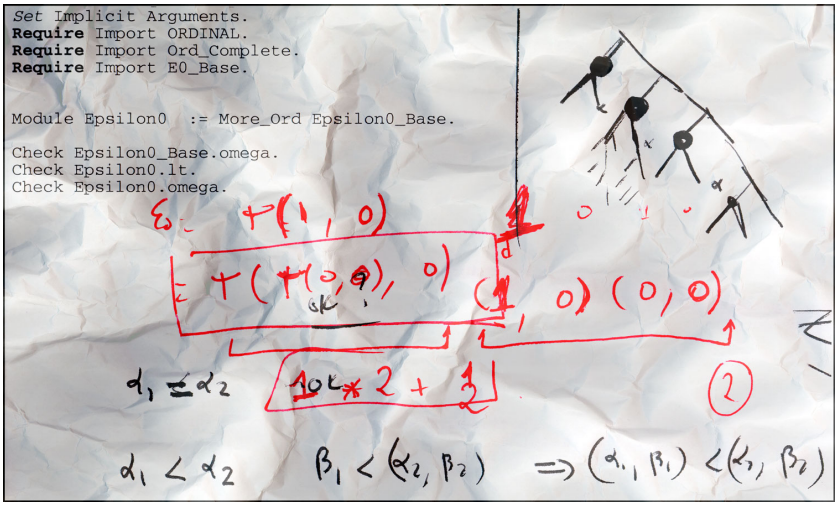

Figure 8.1: Veblen normal form

8.3 How Big is Γ_0?

Let us define a strict order on type `T2`. The following definition is an adaptation of Schütte's, taking into account the multiplications by a natural number (inspired by [MV05], and also present in `T1`).

```
Inductive lt : T2 -> T2 -> Prop :=
| (* 1 *)
 lt_1 : forall alpha beta n gamma,  zero t2< gcons alpha beta n gamma
| (* 2 *)
 lt_2 : forall alpha1 alpha2 beta1 beta2 n1 n2 gamma1 gamma2,
                alpha1 t2< alpha2 ->
                beta1 t2< gcons alpha2 beta2 0 zero ->
               gcons alpha1 beta1 n1 gamma1 t2<
               gcons alpha2 beta2 n2 gamma2
| (* 3 *)
 lt_3 : forall alpha1  beta1 beta2 n1 n2 gamma1 gamma2,
               beta1 t2< beta2 ->
               gcons alpha1 beta1 n1 gamma1 t2<
               gcons alpha1 beta2 n2 gamma2

| (* 4 *)
 lt_4 : forall alpha1 alpha2 beta1 beta2 n1 n2 gamma1 gamma2,
               alpha2 t2< alpha1 ->
               [alpha1, beta1] t2< beta2 ->
               gcons alpha1 beta1 n1 gamma1 t2<
               gcons alpha2 beta2 n2 gamma2

| (* 5 *)
lt_5 : forall alpha1 alpha2 beta1 n1 n2 gamma1 gamma2,
               alpha2 t2< alpha1 ->
               gcons alpha1 beta1 n1 gamma1 t2<
               gcons alpha2  [alpha1, beta1] n2 gamma2
```

```
| (* 6 *)
lt_6 : forall alpha1 beta1  n1  n2 gamma1 gamma2,  (n1 < n2)%nat ->
                                    gcons alpha1 beta1 n1 gamma1 t2<
                                    gcons alpha1 beta1 n2 gamma2

| (* 7 *)
  lt_7 : forall alpha1 beta1 n1   gamma1 gamma2,  gamma1 t2< gamma2 ->
                                       gcons alpha1 beta1 n1 gamma1 t2<
                                       gcons alpha1 beta1 n1 gamma2
where  "o1 t2< o2" := (lt o1 o2): T2_scope.

Hint Constructors lt : T2.
```

Seven constructors! In order to get accustomed with this definition, let us look at a small set of examples, covering all the constructors of `lt`.

8.3.1 Examples

Proof of $0 < \epsilon_0$

```
Example Ex1: 0 t2< epsilon0.
Proof.  constructor 1. Qed.
```

Proof of $\omega < \epsilon_0$

```
Example Ex2: omega t2< epsilon0.
Proof. info_auto with T2. (* uses lt_1 and lt_2 *) Qed.
```

Proof of $\psi(\omega, 8) \times 13 + 56 < \psi(\omega, 8) \times 13 + 57$

```
Example Ex3: gcons omega 8 12 56 t2<  gcons omega 8 12 57.
Proof.
  constructor 7; constructor 6; auto with arith.
Qed.
```

Proof of $\epsilon_0 < \psi(2, 1)$

```
Example Ex4: epsilon0 t2< [2,1].
Proof.
   constructor 2; auto with T2.
   - constructor 6; auto with arith.
Qed.
```

Proof of $\psi(2, 1) < \psi(2, 3)$

```
Example Ex5 : [2,1] t2< [2,3].
Proof.
  constructor 3; auto with T2.
```

```
  - constructor 6; auto with arith.
Qed.
```

Proof of $\psi(1,0) \times 13 + \omega < \psi(0, \psi(2,1))$

```
Example Ex6 : gcons 1 0 12 omega t2< [0,[2,1]].
Proof.
  constructor 4.
  - constructor 1.
  - constructor 2.
    + constructor 6; auto with arith.
    + constructor 1.
Qed.
```

Proof of $\psi(2,1) \times 43 + \epsilon_0 < \psi(1, \psi(2,1))$

```
Example Ex7 : gcons 2 1 42 epsilon0 t2< [1, [2,1]].
Proof.
 constructor 5.
 constructor 6; auto with arith.
Qed.
```

Project 8.4 Write a tactic that solves automatically goals of the form (α `t2<` β), where α and β are closed terms of type `T2`.

8.4 Veblen Normal Forms

Definition 8.1 *A term of the form* $\psi(\alpha_1, \beta_1) \times n_1 + \psi(\alpha_2, \beta_2) \times n_2 + \cdots + \psi(\alpha_k, \beta_k) \times n_k$ *is said to be in* [Veblen] normal form *if for every* $i < n$, $\psi(\alpha_i, \beta_i) < \psi(\alpha_{i+1}, \beta_{i+1})$, *all the* α_i *and* β_i *are in normal form, and all the* n_i *are strictly positive integers.*

```
Inductive nf : T2 -> Prop :=
| zero_nf : nf zero
| single_nf : forall a b n, nf a ->  nf b -> nf (gcons a b n zero)
| gcons_nf : forall a b n a' b' n' c',
                      [a', b'] t2< [a, b]  ->
                      nf a -> nf b ->
                      nf(gcons a' b' n' c')->
                      nf(gcons a b n (gcons a' b' n' c')).
```

Let us look at some positive examples (we have to prove some inversion lemmas before proving counter-examples).

```
Lemma  nf_fin i : nf (fin i).
Proof.
  destruct i.
  - auto with T2.
  - constructor 2; auto with T2.
```

```
Qed.

Lemma nf_omega : nf omega.
Proof.  compute; auto with T2. Qed.

Lemma nf_epsilon0 : nf epsilon0.
Proof. constructor 2; auto with T2. Qed.

Lemma nf_epsilon : forall alpha, nf alpha -> nf (epsilon alpha).
Proof. compute; auto with T2. Qed.

Example Ex8: nf (gcons 2 1 42 epsilon0).
Proof.
  constructor 3; auto with T2.
  - apply Ex4.
  - apply nf_fin.
  - apply nf_fin.
Qed.
```

8.4.1 Length of a Term

The notion of *term length* is introduced by Schütte as a helper for proving (at least) the *trichotomy* property and transitivity of the strict order `lt` on `T2`. These properties are proved by induction on length.

```
Fixpoint nbterms (t:T2) : nat :=
  match t with zero => 0
             | gcons a b n v => (S n) + nbterms v
  end.

Fixpoint t2_length (t:T2) : nat :=
  match t  with
    zero => 0
  | gcons a b n v =>
       nbterms (gcons a b n v) +
      2 * (Max.max (t2_length a)
                               (Max.max (t2_length b)
                                                  (t2_length_aux v)))
  end
with t2_length_aux (t:T2) : nat :=
 match t with
 | zero => 0
  | gcons a b n v =>
           Max.max (t2_length a)
                           (Max.max (t2_length b) (t2_length_aux v))
 end.
```

```
Compute t2_length (gcons 2 1 42 epsilon0).
```

```
= 48 : nat
```

8.4.2 Trichotomy

Trichotomy is another name for the well-known property of decidable total ordering (like Standard Library's `Compare_dec.lt_eq_lt_dec`).

We first prove by induction on l the following lemma:

From `Gamma0.Gamma0`

```
Lemma tricho_aux (l: nat) : forall t t' :T2,
      t2_length t + t2_length t' < l  ->
      {t t2< t'} + {t = t'} + {t' t2<  t}.
```

Then we get our version of `lt_eq_lt_dec`, and derive a comparison function;

```
Definition lt_eq_lt_dec (t t': T2) : {t t2< t'}+{t = t'}+{t' t2<  t}.
Proof.
  eapply tricho_aux.
  eapply lt_n_Sn.
Defined.

Definition compare (t1 t2 : T2) : comparison :=
  match lt_eq_lt_dec t1 t2 with
  | inleft (left _) => Lt
  | inleft (right _) => Eq
  | inright _ => Gt
  end.
```

With the help of `compare`, we get a boolean version of `nf` (being in Veblen normal form).

```
Fixpoint nfb (alpha : T2) : bool :=
  match alpha with
    zero => true
  | gcons a b n zero => andb (nfb a) (nfb b)
  | gcons a b n ((gcons a' b' n' c') as c) =>
    match compare [a', b'] [a, b] with
           Lt => andb (nfb a) (andb (nfb b) (nfb c))
           | _ => false
           end
end.
```

```
Compute compare (gcons 2 1 42 epsilon0) [2,2].
```

```
   = Lt
     : comparison
```

```
Compute nfb  (gcons 2 1 42 epsilon0).
```

```
   = true
     : bool
```

```
Compute nfb (gcons 2 1 42 (gcons 2 2 4 epsilon0)).
```

```
   = false
     : bool
```

Remark 8.1 The connexion between the predicate `nf` and the relation `lt` on one part, and the functions `nfb` and `compare` on the other, is expressed by the following lemmas:

```
Lemma nfb_equiv gamma : nfb gamma = true <-> nf gamma.

Lemma compare_correct alpha beta :
  CompareSpec (alpha = beta) (lt alpha beta) (lt beta alpha)
              (compare alpha beta).
```

The function `compare` helps to make easier proofs of inequalities of closed terms of type `T2`.

First, we prove a lemma:

```
Lemma compare_Lt : forall alpha beta, compare alpha beta = Lt ->
                                          alpha t2< beta.
Proof.
  intros alpha beta; destruct (compare_correct alpha beta);
    trivial; discriminate.
Qed.
```

Then, we give another version of the proof of Sect. 8.3.1 on page 151.

```
Example Ex6 : gcons 1 0 12 omega t2< [0,[2,1]].
Proof. now apply compare_Lt. Qed.
```

8.5 Main Functions on `T2`

8.5.1 Successor

The successor function is defined by structural recursion.
From `Gamma0.T2`

```
Fixpoint succ (a:T2) : T2 :=
 match a with zero => one
            | gcons zero zero n c => fin (S (S n))
            | gcons a b n c => gcons a b n (succ c)
 end.
```

8.5.2 Addition

Like for Cantor normal forms (see Sect. 4.1.7.2), the definition of addition in `T2` requires comparison between ordinal terms.

```
Fixpoint plus (t1 t2 : T2) {struct t1}:T2 :=
  match t1,t2 with
  |  zero, y  => y
  |  x, zero => x
  |  gcons a b n c, gcons a' b' n' c' =>
     (match compare (gcons a b 0 zero)
                    (gcons a' b' 0 zero) with
      | Lt => gcons a' b' n' c'
      | Gt => gcons a b n (c + gcons a' b' n' c')
      | Eq => gcons a b (S(n+n')) c'
      end)
  end
where "alpha + beta" := (plus alpha beta): T2_scope.
```

```
Example Ex7 : 3 + epsilon0 = epsilon0.
Proof. trivial. Qed.
```

8.5.3 The Veblen Function ϕ

The enumeration function of critical ordinals, presented in Sect. 7.8 on page 142, is recursively defined in type T2.

```
Definition  phi (alpha beta : T2) : T2 :=
  match beta with zero => [alpha, beta]
              | [b1, b2] =>
                (match compare alpha b1
                 with Datatypes.Lt => [b1, b2 ]
                 | _ => [alpha,[b1, b2]]
                 end)
              | gcons b1 b2 0 (gcons zero zero  n zero) =>
                (match compare alpha b1
                 with  Datatypes.Lt =>
                       [alpha, (gcons b1 b2 0 (fin n))]
                 | _ =>  [alpha, (gcons b1 b2 0 (fin (S n)))]
                 end)
              | any_beta => [alpha, any_beta]
  end.
```

Despite its complexity, the function phi is well adapted to proofs by simplification or computation.

```
Example Ex8:  phi 1 (succ epsilon0) = [1, [1,0] + 1].
Proof. reflexivity. Qed.
```

```
(**  All epsilons are fixpoints of phi 0 *)

Theorem epsilon_fxp : forall beta, phi zero (epsilon beta) =
                                   epsilon beta.
Proof. reflexivity. Qed.

Theorem epsilon0_fxp : epsilon0 = phi zero epsilon0.
Proof. apply epsilon_fxp. Qed.
```

The relation between the constructor ψ and the function ϕ is studied in [Sch77], and partially implemented in this development. *Please contribute!*

For instance, the following theorem states that, if γ is the sum of a limit ordinal β and a finite ordinal n, and β is a fixpoint of $\phi(\alpha)$, then $\psi(\alpha, \gamma) = \phi_\alpha(\gamma + 1)$.

```
phi_psi :
forall (alpha : T2) [beta gamma : T2] [n : nat],
nf gamma ->
limit_plus_fin beta n gamma ->
phi alpha beta = beta -> [alpha, gamma] = phi alpha (succ gamma)
```

```
Example Ex9 : [zero, epsilon 2 + 4] = phi 0 (epsilon 2 + 5).
Proof. trivial. Qed.
```

On the other hand, ϕ can be expressed in terms of ψ.

```
phi_of_psi:
  forall a b1 b2 : T2,
  phi a [b1, b2] = (if lt_ge_dec a b1 then [b1, b2] else [a, [b1, b2]])
```

```
Example Ex10 : phi omega [epsilon0, 5] = [epsilon0, 5].
Proof. reflexivity. Qed.
```

Project 8.5 Please study a way to pretty print ordinal terms in Veblen normal form (see Section 4.1.8 on page 77).

8.6 An Ordinal Notation for Γ_0

In order to consider type `T2` as an ordinal notation, we have to build an instance of class `ON` (See Definition page 46).

First, we define a type that contains only terms in Veblen normal form, and redefine `lt` and `compare` by delegation (see for comparison the construction of type `E0` in Sect. 4.1.5.1 on page 73).

```
Module G0.

Class G0 := mkg0 {vnf : T2; vnf_ok : nfb vnf}.

Definition lt (alpha beta : G0) := T2.lt (@vnf alpha) (@vnf beta).

Definition compare alpha beta := Gamma0.compare (@vnf alpha) (@vnf beta).
```

Then, we prove that `lt` is a well-founded strict order and that the function `compare` is correct.

```
Instance lt_sto : StrictOrder lt.

Lemma lt_wf : well_founded lt.

Lemma compare_correct alpha beta :
  CompareSpec (alpha = beta) (lt alpha beta) (lt beta alpha)
              (compare alpha beta).

Instance Gamma0: ON lt  compare.
Proof.
  split.
  - apply lt_sto.
  - apply lt_wf.
  - apply compare_correct.
Qed.
```

Remark 8.2 The proof of `lt_wf` has been written by Évelyne Contejean, using her library on the recursive path ordering (see also remark 4.2 on page 81).

Project 8.6 Prove that `Epsilon0` (page 81) is a sub-notation system of `Gamma0`.

Prove that the implemantations of `succ`, `+`, ϕ_0, etc. are compatible in both notation systems.

Note that a function `T1_inj` from `T1` to `T2` has already been defined. It may help to complete the task.

From `Gamma0.T2`

```
(* injection from T1 *)

Fixpoint T1_to_T2 (alpha :T1) : T2 :=
  match alpha  with
  | T1.zero => zero
  | T1.ocons a n b => gcons zero (T1_to_T2 a) n (T1_to_T2 b)
  end.
```

Project 8.7 Prove that the notation system `Gamma0` is a correct implementation of the segment $[0, \Gamma_0)$ of the set of countable ordinals.

Chapter 9

Appendices

Bibliography

[Bau08] Andrej Bauer. The hydra game. https://github.com/andrejbauer/hydra, 2008.

[BC04] Yves Bertot and Pierre Castéran. *Interactive Theorem Proving and Program Development. Coq'Art: The Calculus of Inductive Constructions*. Springer, 2004. `http://www.labri.fr/perso/casteran/CoqArt/index.html`.

[Bur75] William H. Burge. *Recursive programming techniques / William H. Burge*. Addison-Wesley Pub. Co Reading, Mass, 1975.

[BW85] Wilfried Bucholz and Stan Wainer. Provably computable functions and the fast growing hierarchy. In Stephen G. Simpson, editor, *Proceedings of the AMS-IMS-SIAM Joint Summer Research Conference*, 1985.

[Can55] Georg Cantor. *Contributions to the Founding of the Theory of Transfinite Numbers*. Courier Corporation, 1955.

[Cas07] Pierre Castéran. Utilisation en Coq de l'opérateur de description. In *Actes des Journées Francophones des Langages Applicatifs*, 2007. http://jfla.inria.fr/2007/actes/index.html.

[CC06] Pierre Castéran and Évelyne Contéjean. On ordinal notations. User Contributions to the Coq Proof Assistant, 2006.

[CCF+10] Evelyne Contejean, Pierre Courtieu, Julien Forest, Andrei Paskevich, Olivier Pons, and Xavier Urbain. AÒ3pat, an approach for certified automated termination proofs. In *2010 ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation*, pages 63–72. ACM, 2010.

[Chl11] Adam Chlipala. *Certified Programming with Dependent Types*. MIT Press, 2011. `http://adam.chlipala.net/cpdt/`.

[CLKK07] Hubert Comon-Lundh, Claude Kirchner, and Hélène Kirchner, editors. *Rewriting, computation and proof : essays dedicated to Jean-Pierre J ouannaud on the occasion of his 60th birthday*. Lecture Notes in Computer Science. Springer, Berlin, New York, 2007.

[Coq] Coq Development Team. The coq proof assistant. https://coq.inria.fr.

[Der82] Nachum Dershowitz. Orderings for term-rewriting systems. *Theoretical Computer Science*, 17(3):279 – 301, 1982.

[DM07] Nachum Dershowitz and Georg Moser. The hydra battle revisited. In Hubert Comon-Lundh, Claude Kirchner, and Hélène Kirchner, editors, *Rewriting, Computation and Proof: Essays Dedicated to Jean-Pierre Jouannaud on the Occasion of His 60th Birthday*, pages 1–27. Springer Berlin Heidelberg, Berlin, Heidelberg, 2007.

[GAA+13] Georges Gonthier, Andrea Asperti, Jeremy Avigad, Yves Bertot, Cyril Cohen, François Garillot, Stéphane Roux, Assia Mahboubi, Russell O'Connor, Sidi Biha, Ioana Pasca, Laurence Rideau, Alexey Solovyev, Enrico Tassi, and Laurent Théry. A machine-checked proof of the odd order theorem. pages 163–179, 07 2013.

[Gal91] Jean H. Gallier. What's so special about Kruskal's theorem and the ordinal Gamma_0? A survey of some results in proof theory. *Ann. Pure Appl. Log.*, 53(3):199–260, 1991.

[Gon08] Georges Gonthier. Formal proof — the four-color theorem. *Notices of the American Mathematical Society*, 55(11), December 2008.

[Goo44] R. L. Goodstein. On the restricted ordinal theorem. *Journal of Symbolic Logic*, 9(2):33–41, 1944.

[Gri13] José Grimm. Implementation of three types of ordinals in Coq. Research Report RR-8407, INRIA, 2013.

[H+15] Thomas Hales et al. A formal proof of the Kepler conjecture. https://arxiv.org/abs/1501.02155, 2015.

[Hue97] Gérard Huet. The zipper. *J. Funct. Program.*, 7(5):549–554, September 1997.

[KP82] Laurie Kirby and Jeff Paris. Accessible independence results for Peano arithmetic. *Bulletin of the London Mathematical Society*, 14:725–731, 1982.

[KS81] Jussi Ketonen and Robert Solovay. Rapidly growing Ramsey functions. *Annals of Mathematics*, 113(2):267–314, 1981.

[MT] Assia Mahboubi and Enrico Tassi. Mathematical components. https://math-comp.github.io/mcb/.

[MV05] Panagiotis Manolios and Daron Vroon. Ordinal arithmetic: Algorithms and mechanization. *Journal of Automated Reasoning*, 34(4):387–423, May 2005.

[P+] Benjamin Pierce et al. Software foundations. https://softwarefoundations.cis.upenn.edu/.

[Prő13] Hans Jürgen Prőmel. *Ramsey Theory for Discrete Structures*, chapter Rapidly Growing Ramsey Functions. Springer, Cham, 2013.

[Sch77] Kurt Schutte. *Proof theory / Translation from the German by J. N. Crossley.* Springer-Verlag Berlin ; New York, 1977.

[Sla07] Will Sladek. The Termite and the Tower: Goodstein sequences and provability in pa. `www.uio.no/studier/emner/matnat/ifi/INF5170/v08/undervisningsmateriale/sladekgoodstein.pdf`, 2007.

[SM19] Matthieu Sozeau and Cyprien Mangin. Equations reloaded. *Proceedings of the ACM on Programming Languages*, 3(ICFP):1–29, July 2019.

[Tel00] Gerard Tel. *Introduction to Distributed Algorithms.* Cambridge University Press, 2 edition, 2000.

[Wai70] S. S. Wainer. A classification of the ordinal recursive functions. *Archiv für mathematische Logik und Grundlagenforschung*, 13(3):136–153, Dec 1970.

9.1 Future Work (projects)

This document and the proof scripts are far from being complete.

First, there must be a lot of typos to correct, references and index items to add. Many proofs are too complex and should be simplified, etc.

The following extensions are planned, but help is needed:

- Semi automatic tactics for proving inequalities $\alpha < \beta$, even when α and β are not closed terms.
- Extension to Γ_0 (in Veblen normal form)
- More lemmas about hierarchies of rapidly growing functions, and their relationship with primitive recursive functions and provability in Peano arithmetic (following [KS81, KP82]).
- From Coq's point of view, this development could be used as an illustration of the evolution of the software, every time new libraries and sets of tactics could help to simplify the proofs.

9.2 How to Install the Libraries

- The present distribution has been checked with version 8.11.0 of the Coq proof assistant, with the plug-ins `coq-paramcoq` and `coq-equations`.
- just go into the `theories` directory, and type "`make`"

9.3 Comments on Exercises and Projects

Although we do not plan to include complete solutions to the exercises, we think it would be useful to include comments and hints, and questions/answers from the users. In constrast, "projects" are supposed, once completed, to be included in the repository.

9.4 Index

Table 9.1: Main types and type classes

Name	Description	Page
Battle	type class of battle types	28
E0	(well formed) ordinals below ϵ_0	73
Hvariant	variants for proving termination of hydra battles	38
Hydra	tree-like representation of hydras	19
Hydrae	finite sequences of hydras	19
ON	ordinal notations	46
ON_for	comparison of an ordinal notation with Schütte's model	63
ON_iso	isomorphism of ordinal notations	64
Ord	countable ordinals (after Schütte)	128
ppT1	pretty printed version of T1	77
SubON	comparison of ordinal notations	62
T1	ordinal terms below ϵ_0	68
T2	ordinal terms below Γ_0	148
WO	well orders (Schütte's definition)	128

Table 9.2: Main functions and constants

Name	Gallina	Math	Description	Page
canon	canon alpha i	$\{\alpha\}(i)$	Canonical sequence	91
canonS	canonS alpha i	$\{\alpha\}(i+1)$	Helper for canon	91
F_	F_ alpha n	$F_\alpha(n)$	Wainer's `F` fast growing hierarchy	123
H_	H_ alpha n	$H_\alpha(n)$	Hardy's `H` fast growing hierarchy	117
iterate	iterate f n x	$f^{(n)}(x)$	Functional iteration	31
L_	L_ alpha k	$L_\alpha(k)$	final step of a standard path	114
succ	succ alpha		Successor	75, 132 …
zero:Ord		0	The least ordinal (Schütte's model)	130

Table 9.3: Main predicates

Name	Gallina	Math	Description	Page
`lt : T1->T1->Prop`	lt alpha beta	$\alpha < \beta$	strict order on type `T1` [1]	71
`LT: T1->T1->Prop`	alpha o< beta	$\alpha < \beta$	strict order on type `T1` [2]	73
`Lt : E0->E0->Prop`	alpha o< beta	$\alpha < \beta$	strict order on type `E0` [3]	74
`nf: T1->Prop`	`nf alpha`		alpha is in Cantor normal form	72

[1] This order is total, but not well-founded, because of not well formed terms.
[2] Restriction of `lt` to terms in normal form; this order is partial, but well-founded.
[3] This order is total *and* well-founded.

Table 9.4: Infix notations

Name	Gallina	Math	Description	Page
`on_lt`	`alpha o< beta`	$\alpha < \beta$	ordinal inequality [1]	47
`on_le`	`alpha o<= beta`	$\alpha \leq \beta$	ordinal inequality	47
`plus`	`alpha + beta`	$\alpha + \beta$	ordinal addition	76, …
`oplus`	`alpha o+ beta`	$\alpha \oplus \beta$	Hessenberg sum	82
`round`	`h -1-> h'`		one round of a battle	26
`rounds`	`h -+-> h'`		one or more rounds of a battle	27
`round_star`	`h -*-> h'`		any number of rounds of a battle	27

[1] Some notations may belong to several scopes. For instance, "`o<`" is bound in `ON_scope`, `E0_scope`, `t1_scope`, etc., and locally in several libraries.

Table 9.5: Other notations

Name	Gallina	Math	Description	Page
F	`F` n	n	The n-th finite ordinal	69, 134
FS	`FS` n	$n+1$	The $n+1$-th finite ordinal [2]	69
omega	`omega`	ω	the first infinite ordinal	135, 74, 69, …
phi0	`phi0 alpha`	$\phi_0(\alpha)$, ω^α	exponential of base ω	69

[2] Note that there exist also various coercions from `nat` to types of ordinal. Depending on the current scope and Coq's syntactic analysis algorithm, `F` may be left implicit.

Index