Hydra Battles and Cie

Pierre Castéran
LaBRI, Univ. Bordeaux, CNRS UMR 5800, [1]

August 30, 2021

[1]With contributions by Yves Bertot, Évelyne Contejean, Jérémy Damour, Florian Hatat, Pascal Manoury and Théo Zimmermann. The formalization of primitive recursive functions was originally authored by Russel O'Connor[O'C05].

Contents

Chapter 1

Introduction

1.1 Generalities

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 [The, BC04], HOL [GM93], Isabelle [NPW02], 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 [HAB+17]. 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.

In this document, we present two examples which seem to have the above properties.

- 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 the Internet where you can find tutorials on hydra games, together with simulators you can play with. See, for instance, the blogpost and source code written by Andrej Bauer [Bau08, Bau].

  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].

Let us underline the analogy between hydra battles and interactive theorem proving. Hercules is the user (you!), and hydra's heads are the subgoals: you may think that applying a tactic would solve a subgoal, but it results often in the multiplication of such tasks.

- In the second part, we are interested in computing x^n with the least number of multiplications as possible. We use the notion of *addition chains* [Bra39, BB87], to generate efficient certified exponentiation functions.

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 the Coq website [The]. The 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.

1.1.1 Documenting theories with Alectryon

Quotations of Coq source and answers are progressively replaced from copy-pasted *verbatim* to automatically generated *LaTeX* blocks, using Clément Pit-Claudel's *Alectryon* tool [PC20, PC]. Many thanks to Jérémy Damour and Théo Zimmermann who designed tools for maintaining consistency between the always evolving Coq modules and documentation written in *LaTeX*.

Besides the guarantee of consistency between theories and documentation, we hope to give a corpus for experimenting new ways of documenting the implementation of non-trivial mathematics on a proof assistant.

At present, this document is a hybrid version: Chapter 2 on page 19 to 9 on page 167 are fully adapted to *Alectryon*. The rest of the document (Chapter 10 on page 193 is still to be rewritten.

1.1.2 Trust in our proofs

Unlike mathematical literature, where definitions and proofs are spread out over many articles and books, the whole proof is now inside your computer. It is composed from 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 favorite 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

illustrating some tactic of proof pattern, for instance). We use Coq's module system to make several proofs of a given theorem co-exist in our libraries (see also Sect 1.1.8 on page 11). 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 (also proven) 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 44).

2. There is no measure into the interval[1] $[0, \omega^2)$ for proving the termination of all hydra battles (Sect. 3.8.3 on page 60).

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 102).

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 93, 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 of our development is axiom-free, and respects the rules of intuitionistic logic. Note that we also use the `Equations` plug-in [SM19] in the definition 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 129).

[1] We use the notation $[a, b)$ for denoting the set of ordinals greater or equal than a and strictly less than b.

```
About F_zero_eqn.
```

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

F_zero_eqn is not universe polymorphic
Arguments F_zero_eqn _%nat_scope
F_zero_eqn is opaque
Expands to: Constant
hydras.Epsilon0.F_alpha.F_zero_eqn
```

```
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 Typographical Conventions

1.1.5.1 Using Alectryon

Alectryon's *LaTeX* mode cannot be mistaken for our hand-made quotations, particularly if you read a color-print. Here is an example from Chapter 9.

```
Fixpoint Ack (m:nat) : nat -> nat :=
  match m with
  | 0 => S
  | S n => fun k =>  iterate (Ack n) (S k) 1
  end.
```

```
Compute Ack 3 2.
```

```
= 29
: nat
```

1.1.5.2 Verbatim quotations

Quotations of Coq source and answers from Chapter 10, as well as definitions from Standard Library and other contributions are displayed as follows.

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

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

Answers from Coq (including subgoals, 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.
```

1.1.6 Remark

In general, we do not include full proof scripts in this document. The only exceptions are very short proofs (*e.g.*, proofs by computation, or by application of automatic tactics). Likewise, we may display only the important steps on a long interactive proof, for instance, in the following lemma (5.5.1.1 on page 106):

```
Lemma Lemma2_6_1 (alpha : T1) :
  nf alpha ->
  forall beta,
    beta t1< alpha  ->
    {n:nat | const_pathS n alpha beta}.
Proof.
  transfinite_induction alpha.
  (* ... *)
Defined.
```

The reader may consult the full proof scripts with Proof General or CoqIDE, for instance.

1.1.7 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 website,
- 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 to our Coq modules may be incorrect if you did not get this `pdf` document by compiling the distribution available at `https://github.com/coq-community/hydra-battles`.

1.1.8 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 are "masked " inside modules called `Bad`, `Bad1`, etc.

```
Module Bad.

  Definition bottom := the_least (Empty_set Ord).

  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.
```

```
exists ! x : Ord, least_member lt (Empty_set Ord) x
```

```
forall a : Ord,
unique (least_member lt (Empty_set Ord)) a ->
a <= zero
```

```
  Abort.

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.

```
Module Alt.

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

End Alt.
```

1.2 How to install the libraries

- The present distribution has been checked with version 8.13.2 of the Coq proof assistant, with the plug-ins `coq-paramcoq`, `coq-equations` and `coq-mathcomp-algebra`.
- Please refer to the README file of the project

1.3 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 contrast, "projects" are supposed, once completed, to be included in the repository.

Please consult the sub-directory `exercises/` of the project (in construction).

1.4 Acknowledgements

Many thanks to Yves Bertot, Évelyne Contejean, Jéremy Damour, Florian Hatat, David Ilcinkas, Pascal Manoury, Karl Palmskog, Clément Pit-Claudel, Sylvain Salvati, Alan Schmitt and Théo Zimmermann for their help on the elaboration of this library and document, and to the members of the *Formal Methods* team and the *Coq working group* at laBRI for their helpful comments on oral presentations of this work.

Many thanks also to the Coq development team, Yves Bertot, and the 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.

1.4.1 Contributions

Yves Bertot made nice optimizations to algorithms presented in Chapter 10. Évelyne Contejean 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.

The formalization of primitive recursive functions was originally a part of Russel O'Connor's work on Gödel's incompleteness theorems [O'C05].

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 share your contributions, for instance using pull requests and issues on GitHub. Thank you in advance!

Part I

Hydras and ordinals

Introduction

In this part, 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 strictly 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" kind of battles [2] 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 literature (including popularization).

Simply put, this document tries to combine 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 mathematics, 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, type and proof.

Difference from Kirby and Paris's work

In [KP82], Kirby and Paris show 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 chose to prove another statement without any reference to PA, by considering a class of proofs indexed by ordinal numbers up to ϵ_0.

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 definitions 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.

[2] 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 20).

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.

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]

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. Contrary to the computer science tradition, we display 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). *For a less formal description of hydras, please see* `https://www.smbc-comics.com/comic/hydra`.

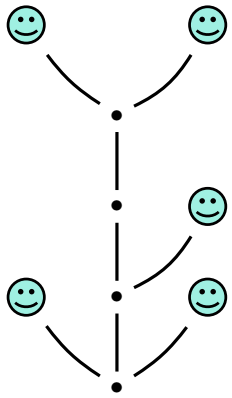


Figure 2.1: The hydra `Hy`

We use a specific vocabulary for talking about hydras. Table 2.2 shows the correspondence 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.

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

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:

- 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 formally study the properties of hydra battles, we prefer to use a mathematical vocabulary, i.e., graphs, relations, functions, etc. Thus, the replication process

[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 22, 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.

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.

2.0.2 Example

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

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

Next, assume Hercules chooses to chop off one of the two highest heads of `Hy'`, for instance the rightmost one. Fig. 2.4 represents the broken segment 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.

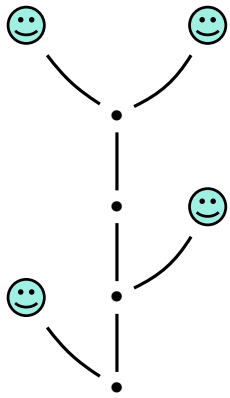


Figure 2.3: `Hy'`: the state of `Hy` after one round

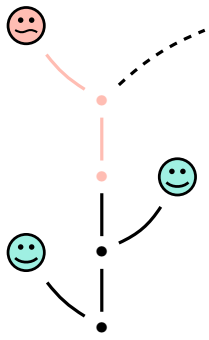


Figure 2.4: A second beheading

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.

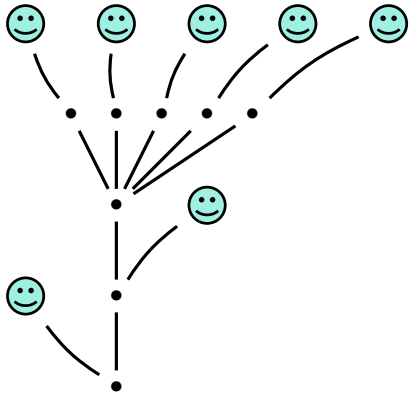


Figure 2.5: `Hy''`: the state of `Hy` after two rounds

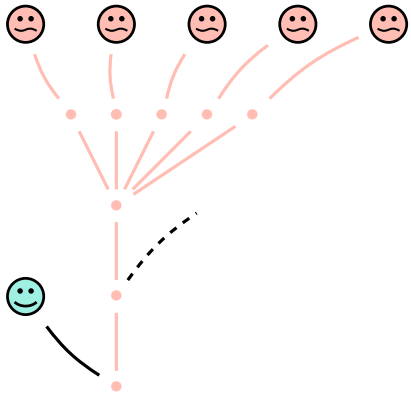


Figure 2.6: A third beheading (wounded part in red)

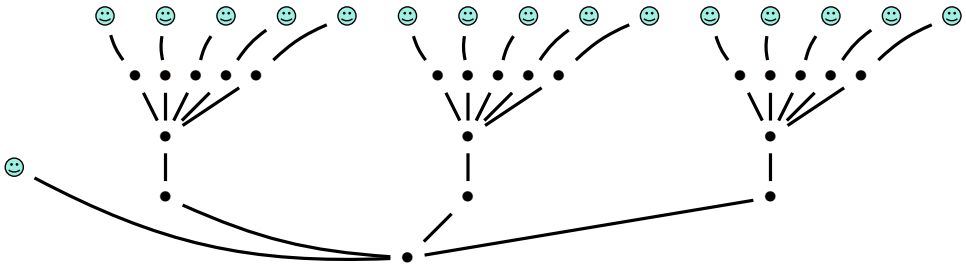


Figure 2.7: The configuration `Hy'''` of `Hy`

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 is 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 [The].

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

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

(** *** Hydra 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 19 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 adjacent copies of the same subtree. 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 equal daughters *)

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

For instance, the hydra `Hy''` of Fig 2.5 on page 22 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`

In order to define 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`:

```
Check Hydra_ind.
```

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

Let 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].
```

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

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

```
    induction s as [| h s'].
```

```
──────────────────────────────────────────────────────────────────────
height head < hsize head
```

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

The first subgoal is trivial.

```
──────────────────────────────────────────────────────────────────────
height head < hsize head
```

```
simpl; auto with arith.
```

Let us look at the second subgoal of the induction.

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

We notice that this subgoal does not contain any hypothesis on the height and size of the hydra `h`. So, it looks hard to prove the conclusion. Let's stop.

```
  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.

From Module Hydra.Hydra_Definitions

```
(** 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.
```

From Module Hydra.Hydra_Examples

```
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)).
```

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

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

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

The first subgoal is easily solvable, using some arithmetic. The second and third ones are almost trivial. We let the reader look at the source.

```
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  :  relation Hydra :=
| 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 19 into the configuration represented in Fig. 2.3.

From Module Hydra.Hydra_Examples

```
Example Hy_1 : R1 Hy Hy'.
Proof.
  repeat constructor.
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)  : relation  Hydrae  :=
| 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 composed of 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)  :  relation Hydra  :=
| 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) :  relation Hydrae :=
     |  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 22). 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.
```

```
R2 4 Hy' Hy''
```

```
    (** move to 2nd sub-hydra (0-based indices) *) r2_up 1.
```

```
R2 4 (hyd2 (hyd1 (hyd2 head head)) head)
  (hyd2 (hyd_mult (hyd1 head) 5) head)
```

```
    (** move to first sub-hydra *)  r2_up 0.
```

```
R2 4 (hyd1 (hyd2 head head)) (hyd_mult (hyd1 head) 5)
```

```
    (** 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 page 32.*

2.2.3 Binary 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 20) 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.
Definition round_star h h' := h = h' \/ round_plus h h'.

Infix "-+->" := round_plus (at level 60).
Infix "-*->" := round_star (at level 60).
```

Exercise 2.3 Prove that if h `-+->` h', then the height of h' is less or equal than the height of 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 22).

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 Hercules's interactive search for 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 round_t := nat -> Hydra -> Hydra -> Prop.

Class Battle :=  {battle_rel : round_t;
                  battle_ok : forall i h h',
                      battle_rel i h h' -> round h h'}.

Arguments battle_rel : clear implicits.
```

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 chose to call *standard*[4] the kind of battles which appear most often in the literature 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.
```

```
i: nat
h, h': Hydra
H: round_n i h h'
──────────────────────────────────────────────────────────────
h -1-> h'
```

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

(** number of steps leading to the hydra's death *)

Definition battle_length B k h l :=
    battle B k h (Nat.pred (k + l)%nat) head.
```

The following property allows us to build battles by composition of smaller ones.

From Module Hydra.Hydra_Lemmas

```
Lemma battle_trans {b:Battle} :
  forall i h j h', battle b i h j h' ->
                   forall k h0,  battle b k h0 i h ->
                                 battle b k h0 j h'.
Proof.
  induction 2.
  - now right with h'0.
  - right with h'';auto.
Qed.
```

[4]This appellation is ours. If there is a better one, we will change it.

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:

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

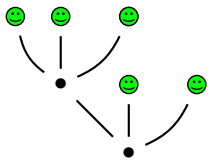


Figure 2.8: The hydra `hinit`

From Module Hydra.BigBattle

```
Notation  h3 := (hyd_mult head 3).
Definition hinit := hyd3 h3  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 any 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 show how we found experimentally the value of the number N. The complete proof is in file `../theories/html/hydras.Hydra.BigBattle.html`.

2.3.1 The beginning of hostilities

During the two first rounds, our hydra loses its two rightmost heads. Figure 2.9 shows the state of the hydra just before the third round.

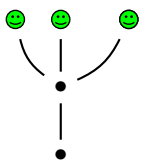


Figure 2.9: The hydra (`hyd1 h3`)

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).
Proof.
  eapply battle_trans with (h := hyd2 h3 head) (i:=1).
  (* ... *)
Qed.
```

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 a few handy abbreviations.

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

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

For instance, the hydra shown in Fig 2.10 is (`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.

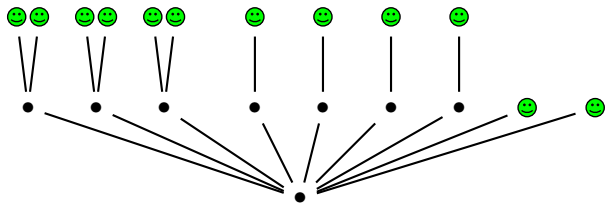


Figure 2.10: The hydra (`hyd 3 4 2`)

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).
Proof.
  left; trivial; right ;  simpl;  left; left.
  split; right; right; left.
Qed.

Lemma L_0_3 : battle standard 0 hinit 3 (hyd 3 0 0).
Proof.
  eapply battle_trans.
  - apply L_2_3.
  - apply L_0_2.
Qed.
```

2.3.3 Testing ...

In order to study *experimentally* the different configurations of the battle, we will use a simple data type 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 configuration of the game. Note that this function is defined only for making experiments and is not "certified". Formal proofs about our battle only start with the lemma `step_battle`, page 38.

```
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 looks 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 battles 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''.

(**  reachability (for i > 0) *)

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 step1.
    + replace (i + S (S c)) with (S i + S c) by ring;  apply IHc.
Qed.


(**   The law that relates two consecutive events with  (nh = 0)  *)


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 and 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.

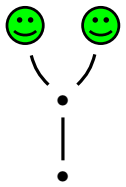


Figure 2.11
The state of the hydra after M rounds.

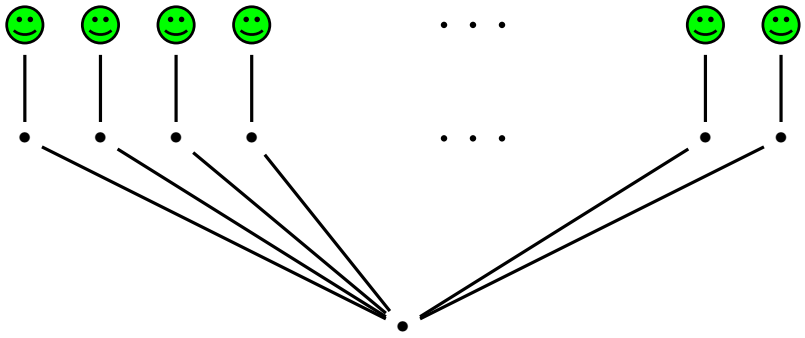


Figure 2.12
The state of the hydra after $M+1$ rounds (with $M+1$ heads).

```
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`).

```
Fixpoint exp2 (n:nat) : nat :=
  match n with
    0 => 1
  | S i => 2 * exp2 i
  end.
```

Using a few 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 |  battle_rel  B i h h'}.
```

The theorems `Alive_free` and `Alive_standard` of the module Hydra.Hydra_Theorems 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

```
(** If the hydra is not reduced to a head, there exists at
    least one head that  Hercules can chop off  *)

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

2.4.2 Termination

The termination of *any* battle 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_rel  B i  h h' -> Lt (m h') (m h)}.
```

Exercise 2.6 Prove that, if there exists some 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 [5].

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

```
Section Impossibility_Proof.

  (** Let us assume there is a variant from Hydra  into nat
      for proving the termination of all hydra battles *)

  Variable m : Hydra -> nat.
  Context (Hvar : @Hvariant _ _ lt_wf free m).
```

We define an injection ι 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 next 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.

[5] 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: forall i,  battle_rel free i big_h 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.
Proof.
  apply (variant_decr  0); auto with hydra.
  discriminate.
Qed.
```

In order to get a contradiction, it suffices to prove the inequality $m(\texttt{big\_h}) \leq m(\texttt{small\_h})$ i.e., $m(\texttt{big\_h} \leq m(\iota(m(\texttt{big\_h})))$.

```
Lemma m_ge : m big_h <= m small_h.
Proof.
  unfold small_h; generalize (m big_h) as i.
```

```
m: Hydra -> nat
Hvar: Hvariant lt_wf free m
iota: nat -> Hydra
big_h: Hydra
small_h: Hydra
──────────────────────────────────────
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 rightmost 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 $S(i) \leq m(\iota(S(i)))$.

We are now ready to complete our impossibility proof.

```
  induction i.
  - auto with arith.
  - apply Lt.le_lt_trans with (m (iota i)).
    (* ... *)
Qed.

Theorem Contradiction : False.
Proof.
 generalize m_lt,  m_ge; intros; lia.
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 formalization 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 $[0, \epsilon_0)$ [6].

[6] We use the mathematical notation $[a, b)$ for the interval $\{x | a \leq x < b\}$.

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 representative of an equivalence class for isomorphisms of totally ordered well-founded sets.

For the computer scientist, ordinals are tools for proving the totality of a given recursive function, or termination of a transition system. *Ordinal arithmetic* provides a set of functions whose properties, like *monotony*, allow to define *variants*, *i.e.* strictly decreasing measures used in proofs of termination.

Let us have a look at Figure 3.1. It presents a few items of a sequence of ordinal numbers, which extends the sequence of natural numbers.

Let us comment some features of this figure:

- The ordinals are listed in a strictly increasing order.
- Dots : "..." stand for infinite sequences of ordinals, not shown for lack of space. For instance, the ordinal 42 is not shown in the first line, but it exists, somewhere 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.

$0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, \ldots$

$\omega, \omega + 1, \omega + 2, \omega + 3, \ldots$

$\omega \times 2, \omega \times 2 + 1, \ldots, \omega \times 3, \omega \times 3 + 1, \ldots, \omega \times 4, \ldots$

$\omega^2, \ldots, \omega^2 \times 42, \ldots, \omega^3, \ldots, \omega^4, \omega^4 + 1, \ldots$

$\omega^\omega, \ldots, \omega^\omega + \omega^7 \times 8, \ldots, \omega^\omega \times 2, \omega^\omega \times 2 + 1, \ldots$

$\omega^{\omega^\omega}, \ldots, \omega^{\omega^\omega} + \omega^\omega \times 42 + \omega^{55} + \omega, \ldots, \omega^{\omega^{\omega+1}}, \omega^{\omega^{\omega+1}} + 1, \ldots$

$\epsilon_0 (= \omega^{\epsilon_0}), \epsilon_0 + 1, \epsilon_0 + 2, \epsilon_0 + 3, \ldots$

$\epsilon_1, \ldots, \epsilon_2, \ldots, \epsilon_\omega, \ldots$

$\Gamma_0, \Gamma_0 + 1, \Gamma_0 + 2, \Gamma_0 + 3, \ldots, \Gamma_0 + \omega, \ldots$

$\ldots$

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

- 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:

[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. In contrast, Coq's standard library contains an inductive definition of a predicate `Acc` which allows us to write constructive proofs of accessibility (See Coq.Init.Wf).

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

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 consider the correspondence between ordinals and sets of ordinals as 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 literature 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 in our development.

- A "mathematical" representation of the set of countable ordinal numbers, after 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* (also called *[ordinal] notation systems*), *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.

It is interesting to compare proofs of a given property (for instance the associativity of addition) both in the computational framework of some ordinal notation, and in the axiomatic model of Schütte.

3.3 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.

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 a bigger ordinal notation or to Schütte's model.

3.3.1 A class for ordinal notations

From the Coq user's point of view, an ordinal notation is a structure allowing to compare two ordinals by computation, and proving by well-founded induction.

3.3.1.1 The `Comparable` class

The following class, contributed by Jérémy Damour and Théo Zimmermann, allows us to apply generic lemmas and tactics about decidable strict orders. The correctness of the comparison 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 Module Prelude.Comparable

```
Class Comparable {A:Type}
  (lt: relation A)
  (compare : A -> A -> comparison) :=
  {
  sto :> StrictOrder lt;
  compare_correct: forall a b,
      CompareSpec (a = b) (lt a b) (lt b a) (compare a b);
  }.
```

3.3.1.2 The `ON` class

The following class definition, parameterized with a type A, a binary relation `lt` on A, specifies that `lt` is a well-founded strict order, provided with a correct comparison function.

From Library OrdinalNotations.ON_Generic

```
Class ON {A:Type}(lt: relation A)
      (compare: A -> A -> comparison)  :=
  {
  comp :> Comparable lt compare;
  wf : well_founded lt;
  }.
```

We give also a few handy definitions and lemmas for any ordinal notation.

```
Section Definitions.

  Context  {A:Type}{lt : relation A}
           {compare : A -> A -> comparison}
           {on : ON lt compare}.

  #[using="All"]
   Definition ON_t := A.

  #[using="All"]
   Definition ON_compare := compare.

  #[using="All"]
   Definition ON_lt := lt.

  #[using="All"]
   Definition ON_le:  relation A := leq lt.

  #[using="All"]
   Definition measure_lt {B : Type} (m : B -> A) : relation B :=
    fun x y =>  ON_lt (m x) (m y).


  #[using="All"]
   Lemma wf_measure {B : Type} (m : B -> A) :
    well_founded (measure_lt m).


  #[using="All"]
   Definition ZeroLimitSucc_dec :=
    forall alpha,
      {Least alpha} +
      {Limit alpha} +
      {beta: A | Successor alpha beta}.

  (** The segment called [O alpha] in Schutte's book *)


  #[using="All"]
   Definition bigO (a: A) : Ensemble A := fun x: A => lt x a.

End Definitions.

Infix "o<" := ON_lt : ON_scope.
Infix "o<=" := ON_le : ON_scope.
Infix "o?=" := ON_compare (at level 70) : ON_scope.
```

Remark 3.1 The infix notations `o<` and `o<=` are 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, however, we still use $<$ and $\leq$ for comparing ordinals.

3.4 Example: the ordinal ω

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

From Library OrdinalNotations.ON_Omega

```
#[global]
 Instance Omega_comp : Comparable Peano.lt  Nat.compare.
Proof.
  split.
  - apply Nat.lt_strorder.
  - apply Nat.compare_spec.
Qed.


#[global]
 Instance Omega : ON Peano.lt Nat.compare.
Proof.
 split.
 - apply Omega_comp.
 - apply Wf_nat.lt_wf.
Qed.

#[local] Open Scope ON_scope.

Compute 6 o?= 9.
```

```
= Lt
: comparison
```

3.5 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.
```

In order to build an instance of `Comparable`, we have to define a correct 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).


#[global] Instance plus_comp : Comparable lt compare.
Proof.
  split.
  - apply lt_strorder.
  - apply compare_correct.
Qed.
```

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

```
Lemma lt_wf : well_founded lt.
Proof. destruct NA, NB.
       apply wf_disjoint_sum; [apply wf | apply wf0].
Qed.
```

Then, we can build an instance of `ON`:

```
#[global] Instance ON_plus : ON lt compare.
Proof.
  split.
  - apply plus_comp.
  - apply lt_wf.
Qed.
```

3.5.1 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). It is also represented by the two first lines of Figure 3.1.

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

```
Instance Omega_plus_Omega: ON _ _ :=  ON_plus Omega Omega.

Definition t := ON_t.

Example ex1 : inl 7 o< inr 0.
Proof.
  apply compare_lt_iff.
```

```
compare Nat.compare Nat.compare (inl 7) (inr 0) = Lt
```

```
  reflexivity.
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:ON_t).


Compute  fin 8 o?= omega.
```

```
= Lt
: comparison
```

```
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 classify its elements into three categories:

- The least ordinal, `(inl 0)`, also known as `(fin 0)`.
- The first infinite ordinal ω.
- The remaining ordinals, either of the form `(inl (S` i`))` or `(inr (S` i`))` (in black on Figure 3.1), called *successor ordinals*.

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 "<=" := (leq 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).

  Definition  Omega_limit
              {sto : StrictOrder lt} (s: nat -> A) (x:A)  :=
    (forall i: nat, s i < x) /\
    (forall y, y  < x -> exists i:nat, y < s i).
```

```
Definition  Omega_limit_s
            `{lt : relation A}
            {sto : StrictOrder lt}
            (s: nat -> A) (x:A) : Type :=
  ((forall i: nat, s i < x) *
   (forall y, y  < x ->  {i:nat | y  < s i}))%type.
```

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

You may start this exercise with the file exercises/ordinals/predSuccUnicity.v.

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

You may start this exercise with the file exercises/ordinals/lt_succ_le.v.

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 define the following function and prove its correctness:

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

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

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

```
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.ON_Omega_plus_omega

```
Definition Zero_limit_succ_dec : ON_Generic.ZeroLimitSucc_dec.
```

3.7 Product of ordinal notations

Let NA and NB be two ordinal notations, on the respective ordered types A and B. The product of NA and NB is considered as the concatenation of B copies of A, ordered by the lexicographic order on $B \times A$.

In Coq, we build an instance of class ON through a sequence of steps as for the sum of ordinal notations.

From Module OrdinalNotations.ON_mult

```
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 := (B * A)%type.
Definition lt : relation t := lexico ltB ltA.


Definition compare (alpha beta: t) : comparison :=
  match compareB (fst alpha) (fst beta) with
  | Eq => compareA (snd alpha) (snd beta)
  | c => c
  end.

#[global] Instance mult_comp:  Comparable lt compare.

#[global] Instance ON_mult: ON lt compare.

End Defs.
```

3.8 The ordinal ω^2

The ordinal ω^2 (also called $\phi_0(2)$, see Chap. 7), is an instance of the multiplication presented in the preceding section.

From Module OrdinalNotations.ON_Omega2

```
Instance Omega2: ON _ _ := ON_mult Omega Omega.

Definition t := ON_t.

Notation omega := (1,0).
Definition zero: t := (0,0).

Definition fin (i:nat) : t := (0,i).
Coercion fin : nat >-> t.
```

3.8.1 Arithmetic of ω^2

3.8.1.1 Successor

The successor of any ordinal is defined by a simple pattern-matching.

```
Notation omega := (1,0).
Definition zero: t := (0,0).

Definition fin (i:nat) : t := (0,i).
Coercion fin : nat >-> t.
```

This function is proved to be correct w.r.t. the `Successor` predicate.

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

Lemma lt_succ_le alpha beta :
  alpha o< beta <-> succ alpha o<= beta.

Lemma lt_succ alpha : alpha o< succ alpha.
Proof.
  destruct alpha; right; cbn; abstract lia.
Qed.
```

3.8.1.2 Addition

We can define on `Omega2` an addition which extends the addition on `nat`. Please note that this operation is not commutative:

```
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 : ON_scope.
Lemma plus_compat (n p: nat) :
  fin  (n + p )%nat =  fin n + fin p.
Proof.
  destruct n; now cbn.
Qed.

Compute 3 + omega.
```

```
= omega
: t
```

```
Compute omega + 3.
```

```
= (1, 3)
: t
```

```
Example non_commutativity_of_plus :  omega + 3 <> 3 + omega.
Proof.  discriminate. Qed.
```

3.8.1.3 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 : ON_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.
Proof.
  destruct alpha as [i j]; exists i; split.
  (* ... *)
```

3.8.2 A proof of termination using ω^2

.

Using the lemma of Sect. 3.3.1.2 on page 51, we can define easily a total function which merges two lists (example contributed by Pascal Manoury).

From Module OrdinalNotations.ON_Omega2

```
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.


End Merge.

Goal forall l, merge nat Nat.leb (nil, l) = l.
```

```
forall l : list nat, merge nat Nat.leb (nil, l) = l
```

```
  intro; now rewrite merge_equation.
Qed.
```

3.8.3 Yet another proof of impossibility

In Sect. 2.4.3 on page 44, we proved that there exists no variant from `Hydra` to `(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 comment 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/ordinals/Hydra/Omega2_Small.v.

3.8.3.1 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 _ _ (ON_Generic.wf (ON:=Omega2)) free m).
```

We follow the same pattern as in Sect. 2.4.3. First, we define an injection ι 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. $\omega \times 3 + 5$.

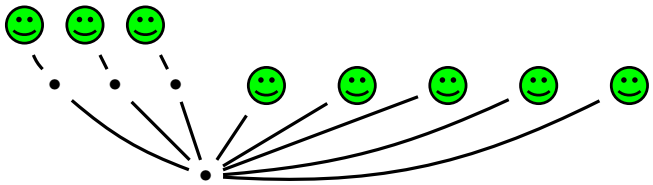


Figure 3.3: The hydra $\iota(\omega \times 3 + 5)$

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[2].

```
  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.

[2]With respect to the measure m.

3.8.3.2 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.
Proof.
  apply m_strict_mono with (1:=Hvar) (2:=big_to_small) .
Qed.
```

3.8.3.3 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 strictly 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 first steps of our proof prepare a well-founded induction on ω^2.

```
Lemma m_ge : m big_h o<= m small_h.
Proof.
  unfold small_h;
  pattern (m big_h);
    apply  well_founded_induction with (R := ON_lt) (1:= wf);
    intros (i,j) IHij.
```

```
m: Hydra -> t
Hvar: Hvariant wf free m
big_h: Hydra
iota: t -> Hydra
small_h: Hydra
i, j: nat
IHij: forall y : t, y o< (i, j) -> y o<= m (iota y)
──────────────────────────────────────────────────────
(i, j) o<= m (iota (i, j))
```

Then a case analysis on i and j allows us to consider three cases :

- $i = j = 0$: the inequality is trivial.
- $i = 1+l, j = 0$ ((i, j) is a limit ordinal): By the induction hypothesis `IHij`, $(l, k) \leq m(\iota(l, k))$ for any k. But (by the rules of the hydra game), $\iota(i, 0)$ is transformed into any $\iota(l, k)$ in one round. Thus $m(\iota(l, k)) < m(\iota(i, 0))$ for any k. Therefore, $(l, k) < m(\iota(i, 0))$ for any k, thus $(i, 0) \leq m(\iota(i, 0))$.

- $j = l + 1$ ((i, j) is a successor). a similar, but simpler case: we apply the induction hypothesis to the pair (i, l).

Please look at the proof script for more details.

```
    (* ... *)

  Qed.
```

3.8.3.4 End of the proof

From `m_ge`, we get `m big_h o<= m small_h = m (iota (m big_h))` . Since $<$ is a strict order (irreflexive and transitive), this inequality is incompatible with the strict inequality `m small_h o< m big_h` (lemma `m_lt`).

From Module Hydra.Omega2_Small

```
  Theorem Impossible : False.
  Proof.
    destruct (StrictOrder_Irreflexive (R:=ON_lt) (m big_h));
      eapply le_lt_trans.
    - 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.

3.9 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

```
(** The type of ordinals less than [n] *)

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

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).

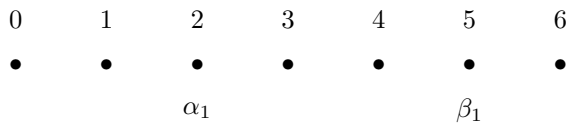


Figure 3.5: The segment $\mathbb{O}_7$

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

Program Example beta1 : t 7 := 5.

Example i1 : lt  alpha1 beta1.
Proof. reflexivity.  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 ; discriminate.
Qed.
```

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

```
1 subgoal (ID 118)

  ============================
  10 <? 10
```

```
  compute.
Abort.
```

In order to build an instance of `ON`, we define a comparison function, and prove its correctiness.

```
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 63). For more information about this pattern, please look at the numerous mailing lists and FAQs on Coq).

Please note 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 alpha1 o< gamma1.
```

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

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 comp n: Comparable (@lt n) compare.

#[global] Instance FinOrd n : ON (@lt n) compare.
```

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 function `bigO` (see Section3.3.1.2 on page 51).

Remark 3.5 There is no interesting arithmetic 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 [MT18]. See also Adam Chlipala's *CPDT* [Chl11] for a thorough study of the use of dependent types.

3.10 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.

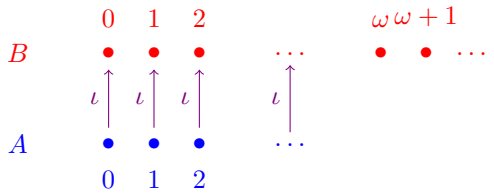


Figure 3.6: A is a sub-segment of B

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

following definition (in ON_Generic).

```
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).

From Module OrdinalNotations.ON_Omega_plus_omega

```
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 $(t\ j)$.

From Module OrdinalNotations.ON_Finite

```
Section Inclusion_ij.

  Variables i j : nat.
  Hypothesis Hij : i < j.

  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.

  Lemma iota_compare_commute alpha beta:
    compare alpha beta =
    compare (iota_ij alpha) (iota_ij beta).

  Lemma iota_mono  : forall alpha beta,
      lt alpha beta <->
      lt (iota_ij alpha) (iota_ij beta).


End Inclusion_ij.
```

Exercise 3.4 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.

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

From Module Schutte.Correctness_E0

```
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.

3.12 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 3.5 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 prove also 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 This exercise is about the non-commutativity of the multiplication of ordinals, reflected in ordinal notations.

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 formally 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.

Note: Like Exercise 3.5, this project corresponds to the [in]equalities $i+\omega = \omega < \omega + i$, for any natural number 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.

3.13 Other ordinal notations

Project 3.7 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 $\alpha = \langle 4,4,2,1,0\rangle$, $\beta = \langle 4,3,3,3,3,2\rangle$, 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 mentioned, 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
| ocons (alpha : T1) (n : nat) (beta : T1) .
```

4.1.2.0.1 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 Library Gamma0.T2, the following type is used to represent ordinals strictly less than Γ_0, in Veblen normal form (see also [Sch77]).

```
Declare Scope T2_scope.
Delimit Scope T2_scope with t2.

Open Scope T2_scope.

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


Notation "[ alpha , beta ]" := (gcons alpha beta 0 zero)
                                 (at level 0): T2_scope.


Definition psi alpha beta  := [alpha, beta].

Definition psi_term alpha :=
  match alpha with zero => zero
                 | gcons a b n c => [a, b]
  end.
```

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)).
```

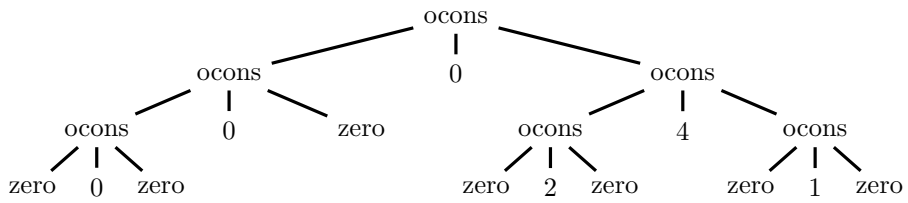


Figure 4.2: The tree-like representation of the ordinal $\omega^\omega + \omega^3 \times 5 + 2$

4.1.2.1.1 Remark For simplicity's sake, we chose to forbid expressions of the form $\omega^\alpha \times 0 + \beta$. Thus, the construction (`ocons` α n β) is intended 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 one := (ocons zero 0 zero).

(** The [(S n)]-th ordinal
 *)

Notation FS n := (ocons zero n zero).

(** the [n]-th (finite) ordinal
 *)

Definition fin (n:nat) := 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).
```

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).
```

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 approximated by the sequence of ω-towers (see also Sect 7.6.3 on page 148).

From Module Epsilon0.T1

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

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

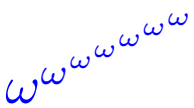


Figure 4.3: The ω-tower of height 7

4.1.4 Pretty-printing ordinals in Cantor normal form

Let us consider again the ordinal α_0 defined in section 4.1.2.1 on page 73 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:

```
Declare Scope ppT1_scope.
Delimit Scope ppT1_scope with pT1.

Inductive ppT1 :=
| PP_fin (_ : nat)
| PP_add (_ _ : ppT1)
| PP_mult (_ : ppT1) (_ : nat)
| PP_exp (_ _ : ppT1)
| PP_omega.


Coercion PP_fin : nat >-> ppT1.

Notation "alpha + beta" := (PP_add alpha beta) : ppT1_scope.

Notation "alpha * n" := (PP_mult alpha n) : ppT1_scope.

Notation "alpha ^ beta" := (PP_exp alpha beta) : ppT1_scope.

Notation _omega := PP_omega.

Check (_omega ^ _omega * 2 + PP_fin 1)%pT1.
```

```
(_omega ^ _omega * 2 + 1)%pT1
     : 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.

4.1.5 Comparison between ordinal terms

In order to compare two terms of type `T1`, we define a recursive function `compare` that maps two ordinal tems 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 beta:T1):comparison :=
  match alpha, beta 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 Nat.compare n n' with
                | Eq => compare b b'
                | comp => comp
                end)
       end)
  end.


Definition lt alpha beta : Prop :=
  compare alpha beta = Lt.
```

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

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

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

Links between the function `compare` and the relations `lt` and `eq` are established through the following lemmas (3.3.1.1 on page 50).

```
#[global] Instance t1_strorder: StrictOrder lt.

#[global] Instance: Comparable lt compare.
```

4.1.5.1 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. Nevertheless, 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
```

4.1.6 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].

```
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 subgoals 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 hydras.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

[1] Ordinal addition is formally defined a little later (page 4.1.8.2)

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

Definition LE := restrict nf (leq lt).
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 alpha :
  alpha t1< one -> alpha = zero.
Proof.
  intros  [H1 [H2 _]]; destruct alpha; auto.
```

```
alpha1: T1
n: nat
alpha2: T1
H1: nf (ocons alpha1 n alpha2)
H2: lt (ocons alpha1 n alpha2) one
──────────────────────────────────────
ocons alpha1 n alpha2 = zero
```

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

4.1.6.1 A sigma-type for ϵ_0

As we noticed in Sect. 4.1.5.1, 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 := mkord {cnf : T1; cnf_ok : nf cnf}.

Arguments cnf : clear implicits.

#[global] Hint Resolve cnf_ok : E0.
```

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 := leq Lt.

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 65).

```
Lemma nf_proof_unicity :
  forall (alpha:T1) (H H': nf alpha), H = H'.
Proof.
  intros; red in H, H'; apply eq_proofs_unicity_on.
  intro y; destruct (bool_dec (nf_b alpha)  y); auto.
Qed.


Corollary E0_eq_iff (alpha beta: E0) :
  alpha = beta <-> cnf alpha = cnf beta.
```

In order to upgrade constants and functions from type `T1` to `E0`, 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`.

```
#[global] Instance Zero : E0 := @mkord zero refl_equal.

#[global] Instance _Omega : E0.
Proof. now exists omega%t1. Defined.

Notation omega  := _Omega.
```

4.1.7 Syntactic definition of limit and successor ordinals

Pattern matching and structural recursion allow us to define boolean characterizations of successor and limit ordinals.

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 limitb 42.
```

```
= false
: bool
```

```
Compute succb 42.
```

```
= true
: bool
```

```
Compute succb omega.
```

```
= false
: 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_lub`, page 97, shows that if α is (syntactically) a limit ordinal, then it is the least upper bound of a strictly increasing sequence of ordinals.

4.1.8 Arithmetic on ϵ_0

4.1.8.1 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 => fin 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 4.1 Prove in Coq that for any ordinal $0 < \alpha < \epsilon_0$, α is a limit if and only if for all $\beta < \alpha$, the interval $[\beta, \alpha)$ is infinite.

You may start this exercise with the file exercises/ordinals/Limit_Infinity.v.

4.1.8.2 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, _  => zero
  |  _, zero => zero
  |  ocons zero n _, ocons zero n' b' =>
      ocons zero (Peano.pred((S n) * (S n'))) b'
  |  ocons a n b, ocons zero n' _ =>
      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.
```

4.1.8.3 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 Ex1 :  42 + omega = omega.
Proof. reflexivity. Qed.

Example Ex2 : omega t1< omega + 42.
Proof. now compute. Qed.
```

```
Example Ex3 : 5 * omega = omega.
Proof. reflexivity. Qed.

Example Ex4 : omega t1<  omega * 5.
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.
```

The function `succ` is related with addition through the following lemma:

```
Lemma succ_is_plus_one (alpha : T1) :  succ alpha = alpha + 1.
Proof.
  induction alpha as [ |a IHa n b IHb]; [trivial |].
  (* ... *)
Qed.
```

4.1.8.4 Arithmetic on type `E0`

We define an addition in type `E0`, since the sum of two terms in normal form is in normal form too.

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

#[global] 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
```

Remark 4.2 In all this development, two representations of ordinals co-exist: ordinal terms (type `T1`, notation scope `t1_scope`, for reasoning on the tree-structure of Cantor normal forms), and ordinal terms *known to be in normal*

form (type `E0`, notation scope `E0_scope`). Looking at the contexts displayed by Coq prevents you from any risk of confusion.

Exercise 4.2 Prove that for any ordinal α : `E0`, $\omega \leq \alpha$ if and only if, for any natural number i, $i + \alpha = \alpha$.

You may start this exercise with the file exercises/ordinals/ge_omega_iff.v.

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 4.3 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.

You may start this exercise with the file exercises/ordinals/T1_ltNotWf.v.

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: T1,  nf alpha -> Acc LT alpha.
  Proof.
    induction alpha as [| beta IHbeta n gamma IHgamma].
    - split.
      inversion 1.
      destruct H2 as [H3 _]. destruct (not_lt_zero H3).
    - split; intros delta Hdelta.
```

```
IHbeta: nf beta -> Acc LT beta
IHgamma: nf gamma -> Acc LT gamma
delta: T1
Hdelta: delta t1< ocons beta n gamma
──────────────────────────────────────
Acc LT delta
```

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` β p' γ', where $p' < n$. Thus, the induction hypotheses `IHbeta` and `IHgamma` are useless for finishing our proof.

```
    Abort.

End First_attempt.
```

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 α is strongly accessible, then, by definition, ω^α is accessible, thus 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_impl 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).

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

```
Lemma Acc_implies_Acc_strong : forall alpha,
    Acc LT  alpha -> Acc_strong alpha.
Proof.
  (*  main induction (on a's accessibility)   *)
  unfold Acc_strong; intros alpha Aalpha; pattern alpha;
  eapply Acc_ind with (R:= LT);[| assumption];
    clear alpha Aalpha; intros alpha Aalpha IHalpha.
```

```
alpha: T1
Aalpha: forall y : T1, y t1< alpha -> Acc LT y
IHalpha: forall y : T1,
         y t1< 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 prove 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 Epsilon0.T1.

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 (by a simple structural induction!).

```
Theorem nf_Acc (alpha : T1): nf alpha -> Acc LT alpha.
Proof.
  induction alpha.
  - intro; apply Acc_zero.
  - intros; eapply Acc_implies_Acc_strong;auto.
     apply IHalpha1; eauto.
```

```
Acc_strong: T1 -> Prop
alpha1: T1
n: nat
alpha2: T1
IHalpha1: nf alpha1 -> Acc LT alpha1
IHalpha2: nf alpha2 -> Acc LT alpha2
H: nf (ocons alpha1 n alpha2)
──────────────────────────────────────
nf alpha1
```

```
      apply nf_inv1 in H; auto.
Qed.

Corollary T1_wf : well_founded LT.


Definition transfinite_recursor := well_founded_induction_type T1_wf.

Check transfinite_recursor.
```

```
transfinite_recursor
     : forall P : T1 -> Type,
       (forall x : T1,
        (forall y : T1, y t1< x -> P y) -> P x) ->
       forall a : T1, P a
```

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

```
Ltac transfinite_induction alpha :=
  pattern alpha; apply transfinite_recursor.
```

Remark 4.3 The alternate proof of well-foundedness using Évelyne Contejean's work on recursive path ordering [Der82, CPU+10] is available in the library Epsilon0.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.

From Module Epsilon0.E0

```
#[global] Instance E0_comp: Comparable Lt compare.
Proof.
  split.
  - apply E0_sto.
  - apply compare_correct.
Qed.

#[global] Instance Epsilon0 : ON Lt compare.
Proof.
 split.
 - apply E0_comp.
 - apply Lt_wf.
Qed.
```

From Module Schutte.Correctness_E0

```
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.

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

Project 4.2 *This exercise is a continuation of Project 3.12 on page 69.* 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.ON_Generic).

```
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 in the development of the type `E0` yet. 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 refinement of `E0` : an ordinal notation for ω^ω

In Module theories/ordinals/OrdinalNotations/OmegaOmega.v, we represent ordinals below ω^ω by list of pairs of natural numbers (with the same coefficient shift as in `T1`). For instance, the ordinal $\omega^4 \times 10 + \omega^3 + \omega + 5$ is represented by the list `(4,9)::(3,0)::(1,0)::(0,4)::nil`.

The usual operations : `succ`, `+`, `*` are simple variants of the same operations in `T1`.

We establish this representation as a *refinement* of the data types we used to represent ordinals less than ϵ_0. Thus, many properties like well-foundedness of $<$ and associativity of $+$, of this ordinal notations have very short proofs.

4.4 A variant for hydra battles

In order to prove the termination of any hydra battle, we try to define a variant mapping hydras to ordinals strictly less than ϵ_0. In order to make such a variant easy to define (for instance by a structural recursion), we introduce a variant of addition, which, contrary to $+$, is commutative and strictly monotonous in both of its arguments. This last property makes it possible to prove that our function is truly a variant for hydra battles (in Sect. 4.4.3 on page 91).

4.4.1 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 literature, 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.

4.4.1.1 Definition of `oplus`

The definition of `oplus` is recursive in both of its arguments and uses 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.

4.4.1.1.1 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.

4.4.1.2 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 a few rewriting lemmas that will help to simplify expressions of the form (`oplus` α β).

A first set of lemmas correspond to the various cases of `oplus`'s definition. They can be proved almost immediately.

```
Lemma oplus_0_r (alpha : T1) : alpha o+ zero = alpha.
Proof.
  destruct alpha; reflexivity.
Qed.

Lemma oplus_0_l (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.).

4.4.2 More theorems on Hessenberg's sum

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

4.4.2.1 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 gamma,
  nf gamma ->
  forall alpha beta,  nf alpha -> nf beta ->
                      T1.lt alpha gamma ->
                      T1.lt beta gamma ->
                      alpha o+ beta = beta o+ alpha.
Proof.
  intros gamma ; transfinite_induction gamma.
  (* ... *)
```

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.
  intros; apply oplus_comm_0 with (T1.succ (max alpha beta));
    trivial.
  (* ... *)
```

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 ->
                   T1.lt a alpha ->
                   T1.lt b alpha -> T1.lt c alpha ->
                   a o+ (b o+ c) = (a o+ b) o+ c.
Proof.
```

```
    intros alpha; transfinite_induction_lt alpha.
    clear alpha ; intros alpha Hrec Halpha.
    (* ... *)

  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 (T1.succ (max alpha (max beta gamma)));
      trivial.
    (* ... *)
```

4.4.2.2 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!

4.4.3 A termination measure for hydra battles

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

From Module Hydra.Hydra_Termination

```
Fixpoint m (h:Hydra) : T1 :=
  match h with head => T1.zero
             | node hs => ms hs
end
with ms (s:Hydrae) :  T1 :=
  match s with  hnil => T1.zero
              | hcons h s' => T1.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.
  induction h using Hydra_rect2
    with (P0 := fun s =>  nf (ms s)).
```

```
Lemma ms_nf : forall s, nf (ms s).
```

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.
  repeat split; auto with T1; now eapply R2_decr_0 with n. (*. none *)

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 termination of all (free) battles.

```
#[global] Instance HVariant : @Hvariant _ _ E0.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

Accessibility 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 chapters.

- There is no variant mapping the type `Hydra` into the interval $[0, \omega^2)$ (section 3.8.3 on page 60), and a fortiori $[0, \omega)$ (section 2.4.3 on page 44).
- There exists a variant which maps the type `Hydra` into the interval $[0, \epsilon_0)$ (theorem `every_battle_terminates`, in section 4.4.3 on page 91).

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 and 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 44 and 3.8.3 on page 60. 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.

From Module Epsilon0.Canon

```
Fixpoint canon alpha (i:nat) : T1 :=
  match alpha with
    zero => zero
  | ocons zero 0 zero  => zero
  | ocons zero (S k) zero  => FS k
  | ocons gamma 0 zero => (match T1.pred gamma with
                             Some gamma' =>
                             match i with 0 => zero
                                     |   S j => ocons gamma' j zero
                             end
                           | None =>
                             ocons (canon gamma i) 0 zero
                           end)

  |  ocons gamma (S n) zero =>
     (match T1.pred gamma with
       Some gamma' =>
       (match i with
         0 =>  ocons gamma n zero
       | S j => ocons gamma n (ocons gamma' j zero)
       end)
      | None => ocons gamma n (ocons (canon gamma  i) 0 zero)
      end)
   |  ocons alpha n beta => ocons alpha n (canon beta i)
end.
```

In the present state of this library, the following specializations of `canon` are still used in some proofs or lemma statements.

```
Definition  canonS alpha (i:nat) : T1 :=
  canon alpha (S i).

Definition  canon0 alpha  : T1 :=
  canon alpha 0.
```

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

```
Section Canon_examples.
Import T1.

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

```
= (_omega ^ 3)%pT1
: ppT1
```

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

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

```
Compute canon (phi0 10) 0.
```

```
= zero
: T1
```

```
Goal canon (omega ^ omega) 10 = phi0 10.
Proof. reflexivity. Qed.

End Canon_examples.
```

5.2.1 Basic properties of canonical sequences

We did not try to prove that our definition truly 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

For instance, the equality $\{\alpha + 1\}(i) = \alpha$ can be proved by structural induction on α.

```
Lemma canon_succ i alpha :
  nf alpha -> canon (succ alpha) i = alpha.
Proof.
  revert i; induction alpha.
```

```
forall i : nat, nf zero -> canon (succ zero) i = zero
```

```
alpha1: T1
n: nat
alpha2: T1
IHalpha1: forall i : nat,
          nf alpha1 -> canon (succ alpha1) i = alpha1
IHalpha2: forall i : nat,
          nf alpha2 -> canon (succ alpha2) i = alpha2
```

```
forall i : nat,
nf (ocons alpha1 n alpha2) ->
canon (succ (ocons alpha1 n alpha2)) i =
ocons alpha1 n alpha2
```

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

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 derivatives in function definitions by transfinite recursion.

```
Lemma canonS_LT i alpha :
  nf alpha -> alpha <> zero ->
  canon alpha (S i) t1<  alpha.
Proof.
  transfinite_induction_lt alpha.
  (* ... *)
Qed.
```

5.2.1.2 Limit ordinals are truly 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 :
  nf lambda ->
  limitb lambda  ->
  forall beta, beta t1< lambda ->
               {i:nat | beta t1< canon lambda (S i)}.
Proof.
  transfinite_induction 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 (canonS lambda) 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.

Hint: You may add to your function a third argument usually called `fuel` for allowing you to give a structurally recursive function (*cf* the post of Guillaume Melquiond on Coq-club (Dec 21, 2020) `https://sympa.inria.fr/sympa/arc/coq-club/2020-12/msg00069.html`). The type `fuel`, an alternative to `nat` is available on Prelude.Fuel).

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 $i < l$, $\gamma_i \neq 0$ [2] 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 $\alpha \underset{s}{\to} \beta$.

We say also that the considered path from α to β *starts at [index]* s_0 *and ends at* s_l.

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.

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

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

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

Definition path alpha s beta := path_to beta s alpha.
```

Remark 5.3 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 `(List.map S` 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}.
```

From the lemma `canonS_LT` 5.2.1.1 on page 97, 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.
```

Exercise 5.3 (continuation of exercise 5.1 on the preceding page) Define a simply typed function for computing a path from α to β.

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
  | zero => head
  | ocons gamma n beta => node (hcons_mult (iota gamma) (S n)
                                           (iotas beta))
  end
with
iotas (alpha : T1) :  Hydrae :=
  match alpha with
  | 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 $\omega^{\omega+2}+\omega^{\omega}\times 2+\omega+1$

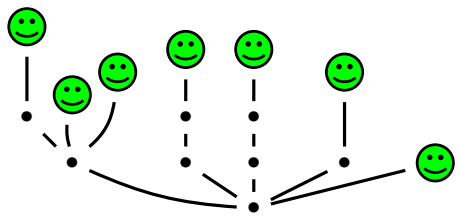


Figure 5.1: The hydra $\iota(\omega^{\omega+2} + \omega^{\omega} \times 2 + \omega + 1)$

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 (canon alpha (S 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_round_plus 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_round_plus 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 44 and 3.8.3 on page 60.

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 {A:Type}{Lt:relation A}
      {Wf: well_founded Lt}{B : Battle}
      {m: Hydra -> A} (Var: Hvariant  Wf  B m)(mu:A):=
  {
  m_bounded: forall h, Lt (m h ) mu
  }.
```

Let us assume there exists such a variant:

```
Section Bounded.

  Context (B: Battle)
          (mu: T1)
          (Hmu: nf mu)
          (m : Hydra -> T1)
          (Var : Hvariant  T1_wf B m)
          (Hy : BoundedVariant  Var mu).

  Hypothesis m_decrease : forall  i h h',
      round_n i h h'-> m h' t1< m h.
```

Remark 5.4 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 44 and 3.8.3 on page 60 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.

  Lemma mu_beta_h : acc_from mu beta_h.
  Proof.
    apply LT_acc_from,  m_bounded.
  Qed.

  Corollary m_ge_generic : m big_h t1<= m small_h.
  Proof.
    apply m_ge_0, nf_m.
  Qed.

End Bounded.
```

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 page 104.

```
Section Impossibility_Proof.

  Context (mu: T1)
          (Hmu: nf mu)
          (m : Hydra -> T1)
          (Var : Hvariant  T1_wf free m)
          (Hy : BoundedVariant Var mu).

  Let big_h := big_h mu.
  Let small_h := small_h mu m.
```

1. The following lemma is an application of `m_ge_generic`, since `free` satisfies trivially the hypothesis `m_decrease` (see page 101).

```
    Lemma m_ge : m big_h t1<= m small_h.
  Proof.
```

```
    apply m_ge_generic with (1 := Hy).
    intros i h h' H; generalize Var; destruct 1.
    apply variant_decr with i.
    intro H0; subst; now apply (head_no_round_n _  _ H).
    exists i; apply H.
  Qed.
```

2. From the hypothesis `Hy`, we have `m big_h t1< mu`

3. By Lemma `LT_to_round_plus`, 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.
  Proof.
    unfold big_h, small_h. apply LT_to_round_plus; auto.
    unfold beta_h. apply (m_bounded big_h); auto.
  Qed.
```

4. From the hypotheses on m, we infer:

```
  Lemma m_lt : m small_h  t1< m big_h.
  Proof. apply m_variant_LT,  big_to_small. Qed.
```

5. From lemmas `m_ge` and `m_lt`, and the irreflexivity of $<$, we get a contradiction.

```
  Fact self_lt_free : m big_h  t1<  m big_h .
  Proof.
    apply LE_LT_trans with (m small_h).
    - apply m_ge.
    - apply m_lt.
  Qed.

  Theorem Impossibility_free : False.
  Proof.  apply (LT_irrefl self_lt_free).  Qed.


End Impossibility_Proof.
```

We have now proved there exists no bounded variant for the class of free battles.

```
Check Impossibility_free.
```

```
Impossibility_free
     : forall (mu : T1) (m : Hydra -> T1)
         (Var : Hvariant T1_wf free m),
       BoundedVariant Var mu -> 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.Paths

```
Inductive standard_pathR (j:nat)( beta : T1):  nat -> T1 -
> Prop :=
  std_1 : forall i alpha,
    alpha <> zero ->
    beta = canon alpha i -> j = i -> i <> 0 ->
    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.


(**  standard path from (i, alpha) to (j, beta) *)

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 $\alpha \overset{*}{\underset{i}{\to}} \beta$ 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 $\alpha \underset{i}{\to} \beta$ for denoting such a path, also called an i-path.

```
Definition const_pathS i :=
  clos_trans_1n T1 (fun alpha beta => alpha <> zero /\
                                      beta = canon alpha (S 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.

From Module Epsilon0.Paths

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

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 $\alpha \underset{n}{\rightarrow} \gamma \underset{n}{\rightarrow} \beta$
- If α is a limit ordinal, we apply the following theorem (numbered `2.4` in Ketonen and Solovay's article).

```
Theorem KS_thm_2_4 (lambda : T1) :
   nf lambda ->
   limitb lambda  ->
   forall i j, (i < j)%nat ->
               const_pathS 0 (canon lambda (S j))
                             (canon lambda (S i)).
Proof.
  transfinite_induction lambda.
  (* ... *)
Qed.
```

We build the following paths :

1. $\alpha \underset{n}{\rightarrow} \{\alpha\}(n)$
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 5.5 `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 corollary `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.
  (* ... *)
Defined.
```

Intuitively, lemma `Lemma2_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  constant_to_standard  (alpha beta : T1) (n : nat):
    nf alpha -> const_pathS n alpha beta ->
    {l : nat | standard_gnaw (S n) alpha l = 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.

Preliminaries

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.
```

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 -> alpha <> zero  ->
                   {j: nat | standard_path (S i) alpha j zero}.
```

Now, 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)
             (Hpa : 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).

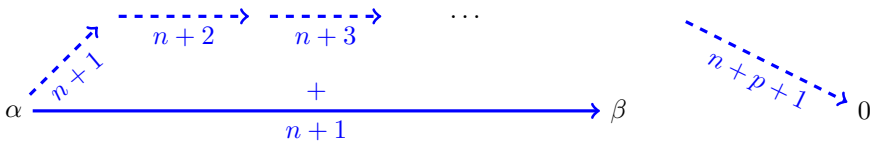


Figure 5.2: A nice proof (1)

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.

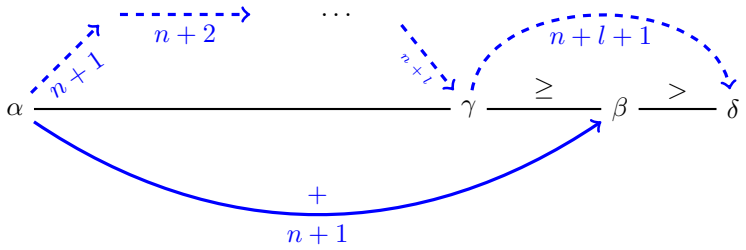


Figure 5.3: A nice proof (2)

- 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 $\beta < \gamma$.

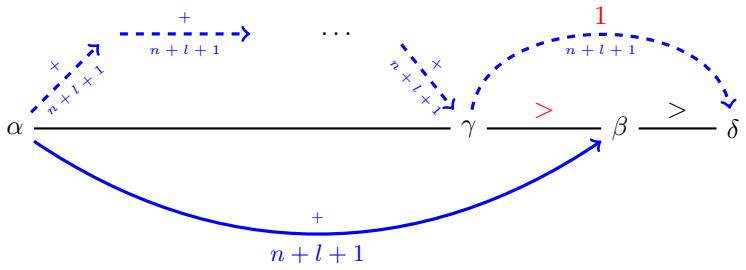


Figure 5.4: A nice proof (3)

The only possible case is thus $\beta = \gamma$, so we have got a standard path from α to β.

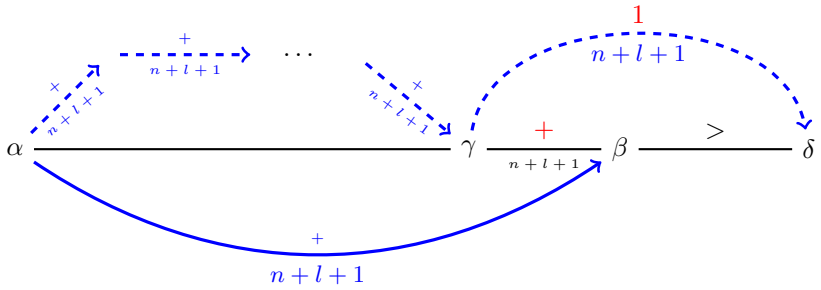


Figure 5.5: A nice proof (4)

```
  Lemma constant_to_standard_0 :
    {l : nat | standard_gnaw (S n) alpha l = beta}.
End Constant_to_standard_Proof.
```

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 Hydra.Epsilon0_Needed_Std. Please note that it has the same global structure as in section5.4.1 Applying the lemmas `Lemma2_6_1` of the module Epsilon0.pathS 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 $\iota(\beta)$, hence the inequality $m(\iota(\beta)) < m(\iota(\alpha))$.

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 (mu: T1)
          (Hmu: nf mu)
          (m : Hydra -> T1)
          (Var : Hvariant  T1_wf standard m)
          (Hy : BoundedVariant Var mu).

  Let big_h := big_h mu.
  Let small_h := small_h mu m.
```

In the same way as for free battles, we import a large inequality from the module 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`$)$.

```
  Fact self_lt_standard : m big_h t1< m big_h.
  Proof.
    apply LE_LT_trans with (m small_h).
    - apply m_ge.
    - apply m_lt.
  Qed.


  Theorem Impossibility_std: False.
  Proof.  apply (LT_irrefl self_lt_standard).  Qed.

End Impossibility_Proof.
```

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 speaking, our implementation *builds*, out of a hypothetical 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 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 135. (please see also the module hydras.Schutte.Correctness_E0).

Thus, we can infer that our theorems can be applied to any well order.

Project 5.1 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

Remark 6.1 Some notations (mainly names of fast-growing functions) of our development may differ slightly from the literature. Although this fact does not affect our proofs, we are preparing a future version where the names *F_alpha*, *f_alpha*, *H_alpha*, etc., are fully consistent with the cited articles.

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 [WB87, Wai70, KS81, Prö13]. 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*, not tests. 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 $s = \langle s_1, s_2, \dots, s_N \rangle$ a finite sequence of strictly positive natural numbers.*

We say that s is α-large *if the sequence $\langle \alpha_0 = \alpha, \dots, \alpha_{i+1} = \{\alpha_i\}(i+1), \dots \rangle$ leads to 0. We say also that s is* minimally α-large *(in short:* α-mlarge*) if s is α-large and every strict prefix of s leads to a non-zero ordinal (*cf *Sect. 5.3.1 on page 98).*

Remark 6.2 Ketonen and Solovay [KS81] consider large finite *sets* of natural numbers, but they are mainly used as sequences. Thus, we chose to represent them explicitely as (sorted) lists.

The following function "gnaws" an ordinal α, following a sequence of indices (ignoring the 0s).

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 (canon alpha (S i)) s'
  end.
```

From Module Epsilon0.Large_Sets

```
Definition largeb (alpha : T1) (s: list nat) :=
  match (gnaw alpha s)
          with zero => true | _ => false end.

Definition large (alpha : T1) (s : list nat) : Prop :=
  largeb alpha s.
```

Minimal large sequences can be directly defined in terms of the predicate `path_to` (5.3.1 on page 98) which already prohibits paths containing non-final `zero`s.

From Module Epsilon0.Large_Sets

```
Definition largeb (alpha : T1) (s: list nat) :=
  match (gnaw alpha s)
          with zero => true | _ => false end.

Definition large (alpha : T1) (s : list nat) : Prop :=
  largeb alpha s.
```

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.

From Module Epsilon0.Large_Sets_Examples

```
Compute pp (gnaw (T1.omega * T1.omega) (interval 6 70)).
```

```
= (_omega * 2 + 56)%pT1
: ppT1
```

Let us try another computation.

```
Compute  (gnaw (T1.omega * T1.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 (T1.omega * T1.omega) (interval 6 510).
Proof with try (auto with arith || discriminate ).
  unfold interval; simpl Peano.minus.
  do 2 rewrite iota_from_unroll; rewrite mlarge_iff  ...
  repeat rewrite not_in_cons ...
Qed.
```

6.2 Length of minimal large sequences

Now, consider any natural number $k > 0$ any ordinal $0 < \alpha < \epsilon_0$. 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, we would like to define 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.

6.2.1 Formal specification

Let $0 < \alpha < \epsilon_0$ be an ordinal term. We consider any function which maps any strictly positive integer k to the number l, where the interval $[k, l)$ is α-mlarge.

Remark 6.3 In [KS81] Ketonen and Solovay consider the least natural number l where the interval $[k, l]$ (l included) is α-large, and call H_α the function which maps k to l. We chose to consider intervals $[l, k)$ instead of $[l, k]$ in order to simplify some statements and proofs in composition lemmas associated with the ordinals of the form $\alpha \times i$ and $\omega^\alpha \times i + \beta$. Clearly, both approaches are related through the equality $L_\alpha(k) = H_\alpha(k) + 1$, for any non-null α and k.

Our specification of the function `L` is as follows:
From Module Epsilon0.Large_Sets

```
Inductive L_spec : T1 -> (nat -> nat) -> Prop :=
  L_spec0 :
    forall f, (forall k, f (S k) = S 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.
```

To do 6.1 *Check if the functions L_α are the same as [KS81]' functions f_alpha (p. 297).*

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 extensionality).

```
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 (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)`.

Useful abstract properties of arithmetic functions are defined in Prelude.Iterates.

```
Definition strict_mono f := forall n p,  n < p -> f n < f p.

Definition strict_mono1 f := forall n p,  0 < n < p -> f n < f p.

Definition dominates_from n g f  := forall p, n <= p -
> f p < g p.

Definition fun_le f g := forall n:nat, f n <=  g n.
Infix "<<=" := fun_le (at level 60).

Definition dominates g f := exists n : nat, dominates_from n g f .
Infix ">>" := dominates (at level 60).

Definition dominates_strong g f  := {n : nat | dominates_from n g f}.
Infix ">>s" := dominates_strong (at level 60).
```

We prove also that the functions we consider are strictly monotonous. The section on successor ordinals has the following structure.

From Module Epsilon0.Large_Sets

```
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 into a first step to $\{\lambda\}(k)$, followed by a path to 0. In the

following section, we assume that there exists a correct function computing $L_{\{\lambda\}(k)}$ 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 (canon lambda (S k)) (f (S k))).

  Remark canon_not_null : forall k,  canon lambda (S k) <> zero.

  Definition  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 obtain a few 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).
Proof.
  induction i.
  (* ... *)
  Qed.
```

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 T1.omega L_omega.
Proof.
  specialize (L_lim_ok T1.omega nf_omega refl_equal L_fin
                       (fun i => L_fin_ok (S i))) ; intro H.
   eapply L_spec_compat with (1:=H).
   intro ; unfold L_lim, L_fin, L_omega; abstract lia.
Qed.
```

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 (T1.omega * i)
          (interval (S k) (2 * (S k))%nat)
          (T1.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  (T1.omega * (S i))
            (interval (S k)
                      (Nat.pred (iterate (fun p =>  S (2 * p)%nat)
                                         (S i)
                                         (S k)))).
```

From Module Epsilon0.Large_Sets

```
Definition L_omega_mult i (x:nat) :=  iterate L_omega i x.

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.
```

Correctness of the function `L_omega_mult` is asserted through the following lemma.

```
Lemma L_omega_mult_ok (i: nat) :
  L_spec (T1.omega * i) (L_omega_mult i).
```

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).
```

```
Compute L_omega_square 8.
```

```
= 2559
: nat
```

```
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 (T1.omega * T1.omega) L_omega_square.
```

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 function associated with α is strictly 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 (T1.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).
Proof.  now  apply iterate_mono. Qed.

Lemma L_phi0_mult_Sle i: S <<=  L_phi0_mult (S i).
Proof.   now apply iterate_Sge. Qed.

End phi0_mult.
```

Let us look now 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}, by diagonalization upon $L_{\omega^2\times i}$.

```
Definition L_omega_cube  := L_lim  L_omega_square_times .
Lemma L_omega_cube_ok : L_spec (T1.phi0 3) L_omega_cube.
Proof.
  unfold L_omega_cube.
  apply L_lim_ok.
  - auto with T1.
  - auto with T1.
  - intro k; simpl canonS; apply L_omega_square_times_ok.
Qed.

Lemma L_omega_cube_eqn i : L_omega_cube i = L_omega_square_times i (S i).
Proof.   reflexivity. Qed.
```

Thus, for instance, $L_{\omega^3}(3) = L_{\omega^2\times 4}(3)$.

```
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 a few 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. To this end, we represent ordinals as inhabitants of the type `E0` of well-formed ordinal terms (see Sect 4.1.6.1 on page 78). So, we define a total function `L_` of type `E0 -> nat -> nat`, by transfinite recursion, considering the usual three cases : $\alpha = 0$, α is a successor, α is a limit ordinal.

6.2.4.1 Definition

From Module L_alpha).

```
From Equations Require Import Equations.
Import RelationClasses Relations.

Instance Olt : WellFounded Lt := Lt_wf.
Global 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.
```

This definition results in a bunch of automatically generated lemmas. For instance:

```
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

L__equation_1 is not universe polymorphic
Arguments L__equation_1 _ _%nat_scope
L__equation_1 is transparent
Expands to: Constant
hydras.Epsilon0.L_alpha.L__equation_1
```

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_eq2 alpha i :
  Succb alpha -> L_ alpha i = L_ (Pred alpha) (S i).
Lemma L_succ_eqn alpha i :
  L_ (Succ alpha) i = L_  alpha (S i).

Lemma L_lim_eqn alpha i :
  Limitb alpha ->
  L_ alpha i = L_ (Canon alpha i) (S i).
```

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.

Lemma L_omega : forall k, L_ omega%e0 k = S (2 * k)%nat.
```

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.

6.2.4.2 Back to hydra battles

Lemma `battle_length_std` of Module Hydra.Battle_length relates the length of standard battles with the functions L_α.

```
Lemma battle_length_std:
  battle_length  standard k (iota (cnf alpha)) (l-k)%nat.
```

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 A variant of the Wainer-Hardy hierarchy

In order to give a feeling on the complexity of the functions L_αs, we compare them with a better known family of functions, the *Wainer-Hardy hierarchy* of fast growing functions, presented for instance in [Prö13].

Remark 6.4 Indeed, the functions presented in this section are a *variant* of the Hardy hierarchy of functions. In the future versions of this development, we will correct the references to the literature. For the time being, we call our functions H'_α in order to underline the difference from "classic" Hardy functions.

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 $k \in \mathbb{N}$.

Remark 6.5 The "classic" definition of the Wainer-Hardy hierarchy differs in the third equation.

- 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 $H_\alpha(k) = H_{(\{\alpha\}(k))}(k)$ for any $k \in \mathbb{N}$.

6.3.1 Definition 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.Hprime

```
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.

(** Paraphrases of the equations for H'_ *)
Lemma H'_eq1 : forall i, H'_ Zero i = i.
Proof.
  intro i; now rewrite H'__equation_1.
Qed.

Lemma H'_eq2 alpha i :
  Succb 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'_succ_eqn  alpha i :
  H'_ (Succ alpha) i = H'_ alpha (S i).
```

6.3.2 First steps of the H' hierarchy

Using rewrite rules from `H'_eq1` to `H'_succ_eqn`, we can explore the functions H'_α for 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 Fin; simpl; autorewrite with H'_rw E0_rw ...
  - intros ;simpl; autorewrite with H'_rw E0_rw ...
    rewrite IHi; abstract 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; abstract 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.
  induction i.
  (* ... *)
Qed.
```

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'_eq3; simpl Canon.
 - ochange  (CanonS  (omega * FinS 1)%e0 k)  (omega + (S k))%e0.
  + rewrite H'_Plus_Fin, H'_omega;  abstract 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:nat)  : 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.
```

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)  :
  forall i k,
    H'_ (Omega_term alpha i) k =
    iterate  (H'_ (phi0 alpha)) (S i) k.
```

3. In particular, we derive a formula for $H'_{\omega^{i+1}}$.

```
Definition H'_succ_fun f k := iterate f (S k) k.

Lemma H'_Phi0_Si : forall i k,
    H'_ (phi0 (S i)) k = iterate H'_succ_fun i (H'_ omega) k.
```

4. 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.
```

6.3.2.4 A numerical example

It looks hard to capture the complexity of this function by looking only at this "exact" formula. Let us consider a simple example: the number $H'_{\omega^3}(3)$.

```
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))).
```

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 with `Ocaml`:

```
# (2.0 ** 64.0 *. 64.0 -. 1.0);;
```

```
- : float = 1.18059162071741113e+21
```

```
Fact F1 : H'_ (phi0 3) 3 = f (f N).
Proof.
  rewrite F0; reflexivity.
Qed.


Lemma F1_simpl :
  H'_ (phi0 3) 3 =
  (exp2 (exp2 (S N) * S N) * (exp2 (S N) * S N) - 1)%nat.


End H'_omega_cube_3.
```

In a more classical writing, this number is displayed as follows:

$$H'_{\omega^3}(3) = 2^{(2^{N+1}\,(N+1))}\,(2^{N+1}\,(N+1)) - 1$$

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 !

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.
```

Using extensionality of the functional `iterate`, we also 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.
Proof.
```

Note that this formula contains two occurrences 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'_α

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

On the contrary, the functions of the H' 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 conjunction 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, using the abstract properties defined in Sect 6.2.2.2 on page 117

```
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.
```

```
alpha: E0
──────────────────────────────────────
well_founded Lt
```

```
alpha: E0
──────────────────────────────────────
forall x : E0, (forall y : E0, y o< x -> P y) -> P x
```

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

End Proof_of_Abstract_Properties.
```

Using a few lemmas *à la* Ketonen-Solovay, we prove that if $\alpha < \beta$, then H'_β eventually dominates H'_α. We let the reader look at its proof (Section `Proof_of_H'_mono_l` of Epsilon0.Hprime).

```
  Theorem H'_dom : dominates_strong (H'_ beta) (H'_ alpha).
```

6.3.4 Comparison between `L_` and `H'_`

By well-founded induction on α, we prove a simple relation between L_α and *H'_alpha*.

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 the H' family of fast growing functions.

From Module Hydra.Hydra_Theorems

```
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.
Proof.
  intros H k  H0; exists (L_ alpha (S k) - k); split.
  - generalize (H'_L_ alpha k); lia.
  - now apply battle_length_std.
Qed.
```

6.4 A variant of the Wainer hierarchy (functions F_α)

Ketonen and Solovay introduce in [KS81] a "trivial" variant of the Wainer hierarchy [WB87, Wai70] of fast growing functions, indexed by ordinals below ϵ_0.

- $F_0(i) = i + 1$
- $F_{\beta+1}(i) = (F_\beta)^{(i+1)}(i)$, where $f^{(i)}$ is the i-th iterate of f.
- $F_\alpha(i) = F_{\{\alpha\}(i)}(i)$ if α is a limit ordinal.

Remark 6.6 The difference with the "classic" Wainer hierarchy f_α ($\alpha < \epsilon_0$) lies in the second equation: $f_{\beta+1}(i) = (f_\beta)^{(i)}(i)$ and not $f_{\beta+1}(i) = (f_\beta)^{(i+1)}(i)$. A module about the classic Wainer hierarchy is in preparation.

A first attempt is to write a definition of F_α by equations, in the same way 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 Epsilon0.F_alpha.

```
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.


(*  F_star (alpha,i) is intended to be the i-
th iterate of F_ alpha *)

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).

Definition F_ alpha i := F_star (alpha, 1) i.
```

It is quite easy to prove that our function `F_` satisfies the equations on page 130.

```
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 i,  F_ 1 i = S (2 * i).

Lemma LF2 : forall i, exp2 i * i < F_ 2 i.
```

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.
```

```
Corollary F_alpha_positive alpha :  forall n, 0 < F_ alpha n.


Theorem F_alpha_dom alpha :
  dominates_from 1 (F_ (Succ alpha)) (F_ alpha).

(** [F_] is not mononotonous in [alpha] in general.
      Nevertheless, this lemma may help (from [KS]) *)


Theorem F_restricted_mono_l alpha :
  forall beta n, Canon_plus n alpha beta ->
                 F_ beta n <= F_ alpha n.
```

As a corollary, we prove the following proposition, p. 284 of [KS81].

If $\beta < \alpha$, F_α dominates F_β.

```
Section Compatibility_F_dominates.

  Variables alpha beta : E0.
  Hypothesis H'_beta_alpha : Lt beta alpha.

  Lemma F_mono_l: dominates (F_ alpha) (F_ beta).

End Compatibility_F_dominates.
```

Exercise 6.1 Prove the following property:

```
Lemma LF3 : dominates_from  2 (F_ 3) (fun  n => iterate exp2 n n).
```

You may start this exercise with the file exercises/ordinals/F_3.v.

Exercise 6.2 Prove that, for any $\alpha \geq 3$ and $n \geq 2$, $F_\alpha(1+n) \geq 2^{F_\alpha(n)}$.
You may start this exercise with the file exercises/ordinals/F_3.v.

Exercise 6.3 It is tempting to prove a simple property of monotony of the function `F_`.

Let $\alpha \leq \beta < \epsilon_0$. For any $n \geq 2$, $F_\alpha(n) \leq F_\beta(n)$.

Prove or disprove this statement.
You may start this exercise with the file exercises/ordinals/is_F_monotonous.v.

Exercise 6.4 Prove that for any $n \geq 2$, Ack $n\,n \leq F_\omega(n)$, where Ack is the Ackermann function. Next, prove that F_α is not primitive recursive, for any $\alpha \geq \omega$ (please see Sect. 9.5 on page 179). On the other hand, please show that for any natural number n, the function F_n is primitive recursive. Thus *F_alpha* is primitive recursive if and only if α is finite.
You may start this exercise with the file exercises/ordinals/F_Omega.v. Properties of the Ackermann function are studied in theories/ordinals/MoreAck/Ack.v and theories/ordinals/MoreAck/AckNotPR.v .

6.5 Conclusion

In Sect. 9.6 on page 187, we prove that the length of hydra-battles (for a given hydra, according to the initial replication factor) is not primitive recursive in general. This proof uses properties of the Ackermann function, and the H'_α, F_α, L_α families of functions.

6.6 A certified catalogue of rapidly growing functions

In this section, we try to present an abstract of the properties of the main variants of fast-growing hierarchies of functions we found in the literature or we had to define as helpers in our proofs.

6.6.1 Ketonen-Solovay's `F_alpha`

This hierarchy is presented p.280 of [KS81] as " a trivial variant of one introduced by Wainer ". Let us recall the equations shown in Sect. 6.4 on page 129.

- $F_0(i) = i + 1$
- $F_{\beta+1}(i) = (F_\beta)^{(i+1)}(i)$, where $f^{(i)}$ is the i-th iterate of f.
- $F_\alpha(i) = F_{\{\alpha\}(i)}(i)$ if α is a limit ordinal.

Note that [KS81] defines also F_{ϵ_0} (by the third equation). Since ϵ_0 is not representable in type `E0`, our translation in Coq does not take F_{ϵ_0} into account.

Several properties of F_α are already presented in Sect. 6.4.

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 hydras.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 : Ensemble Ord := Full_set Ord.
Definition big0 alpha : Ensemble Ord := fun beta =>  beta < alpha.

Global Hint Unfold ordinal : schutte.
```

Schütte's first axiom tells that `lt` is a well order on the set `ordinal` (The class `WO` is defined in Module Schutte.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 decided 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. We would like to call this element zero.

```
Remark R : exists! z : Ord, least_member lt  ordinal z.

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.
```

```
______________________________________
Ord
```

```
Abort.
```

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, The].

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 have to tell Coq that the declared type `Ord` is not empty:

```
Axiom inh_Ord : inhabited Ord.

Instance InH_Ord : InH Ord.
Proof.
  exact inh_Ord.
Qed.
```

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.

From Module Schutte.Well_Orders

```
  Definition least_member  (X:Ensemble M)(a:M) :=
    In X a /\ forall x,  In X x -> Le a x.

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 := 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.
```

```
alpha: Ord
============================
exists ! x : Ord, least_member lt ordinal x
```

```
alpha: Ord
============================
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).
```

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.
```

On the other hand, the following attempt fails, because of the unprovable first subgoal (please notice that the second subgoal is easy to solve !).

```
  Lemma le_bottom_zero : bottom <= zero.
  Proof.
    unfold bottom, the_least, the; apply iota_ind.
```

```
exists ! x : Ord, least_member lt (Empty_set Ord) x
```

```
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 following 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.
  intro alpha; unfold succ, the_least, the; apply iota_spec.
```

```
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 rest of 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.


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 succ_injection (alpha beta : Ord) :
  succ alpha = succ beta -> alpha = beta.

Lemma succ_zero_diff (alpha : Ord): succ alpha <> zero.
```

```
Lemma zero_lt_succ : forall alpha,  zero < succ alpha.

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.

```
Fixpoint finite (i:nat) : Ord :=
  match i with 0 => zero
            | S i => succ (finite i)
  end.

Notation F i := (finite i).

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 upper_bound (M:Type)
                       (D: Ensemble M)
                       (lt: relation M)
                       (X:Ensemble M)
                       (a:M) :=
  forall x, In _ D x ->  In _ X x -> x = a \/ lt  x a.


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. Then we define `omega` as the limit of the sequence of finite ordinals.

```
Definition omega_limit (s:nat->Ord) : Ord
  := |_| (seq_range s).

Definition _omega := omega_limit finite.

Notation omega := (_omega).
```

Among the numerous properties of the ordinal ω, let us quote the following ones (proved in Module `Schutte.Schutte_basics`)

```
Lemma finite_lt_omega (i : nat) : i < omega.


Lemma zero_lt_omega : zero < 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, In 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.
```

```
ordering_function_ex :
forall B : Ensemble Ord,
exists ! S : Ensemble Ord,
  exists f : Ord -> Ord, ordering_function f S B

ordering_function_ex is not universe polymorphic
ordering_function_ex is opaque
Expands to: Constant
hydras.Schutte.Ordering_Functions.ordering_function_ex
```

```
About ordering_function_unicity.
```

```
ordering_function_unicity :
forall [B A1 A2 : Ensemble Ord] [f1 f2 : Ord -> Ord],
ordering_function f1 A1 B ->
ordering_function f2 A2 B -> fun_equiv f1 f2 A1 A2

ordering_function_unicity is not universe polymorphic
Arguments ordering_function_unicity [B A1 A2] [f1
  f2]%function_scope _ _
ordering_function_unicity is opaque
Expands to: Constant
hydras.Schutte.Ordering_Functions.ordering_function_unicity
```

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 13.3 of Schutte's book *)

Theorem ordering_le : forall f A B,
    ordering_function f A B ->
    forall alpha, In A alpha -> alpha  <= f alpha.

(* Theorem 13.5.2 by Schutte *)

About Th_13_5_2.
```

```
Th_13_5_2 :
forall [A B : Ensemble Ord] [f : Ord -> Ord],
ordering_function f A B ->
Closed B -> continuous f A B

Th_13_5_2 is not universe polymorphic
Arguments Th_13_5_2 [A B] [f]%function_scope _ _
Th_13_5_2 is opaque
Expands to: Constant
hydras.Schutte.Ordering_Functions.Th_13_5_2
```

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).

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

As a use-case, let us prove that 0 is a right neutral element of $+$.

```
Lemma alpha_plus_zero (alpha: Ord):   alpha + zero = alpha.
Proof.
  pattern  (plus alpha); apply plus_elim;eauto.
```

```
alpha: Ord
──────────────────────────────────────────────────────────
forall f : Ord -> Ord,
ordering_function f ordinal (ge alpha) ->
f zero = alpha
```

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

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).
```

The following lemmas are not direct applications of `plus_elim`.

```
Theorem plus_assoc (alpha beta gamma : Ord) :
  alpha + (beta + gamma) = (alpha + beta) + gamma.

Lemma finite_plus_infinite (n : nat) (alpha : Ord) :
  omega <= alpha -> n + alpha = alpha.
```

It is interesting 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.

Infix "*" := mult_fin_r : 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 phi0 := _phi0.

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 least_AP : least_member lt AP 1.
Lemma AP_omega : In AP omega.
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.
```

```
Unbounded AP
```

```
Proof.
```

```
Unbounded AP
```

```
  intro x;  pose (H := AP_unbounded_0 x).
```

```
x: Ord
H: x <
   omega_limit
   (fix seq (n : nat) : Ord :=
   match n with
   | 0 => succ x
   | S p => seq p + seq p
   end) /\
   AP
   (omega_limit
   (fix seq (n : nat) : Ord :=
   match n with
   | 0 => succ x
   | S p => seq p + seq p
   end))
------------------------------------------------------------------------
exists y : Ord, In AP y /\ x < y
```

```
 exists (omega_limit
           (fix seq (n : nat) : Ord :=
              match n with
              | 0 => succ x
              | S p => seq p + seq p
              end)).
```

```
x: Ord
H: x <
   omega_limit
   (fix seq (n : nat) : Ord :=
   match n with
   | 0 => succ x
   | S p => seq p + seq p
   end) /\
   AP
   (omega_limit
   (fix seq (n : nat) : Ord :=
   match n with
   | 0 => succ x
   | S p => seq p + seq p
   end))
------------------------------------------------------------------------
In AP
   (omega_limit
      (fix seq (n : nat) : Ord :=
         match n with
         | 0 => succ x
         | S p => seq p + seq p
         end)) /\
x <
omega_limit
   (fix seq (n : nat) : Ord :=
      match n with
      | 0 => succ x
      | S p => seq p + seq p
      end)
```

```
  now destruct H.
Qed.
```

Finally, AP is (topologically) *closed* and ordered by the segment of all countable ordinals.

From Module Schutte.Schutte_basics

```
Definition Closed (B : Ensemble Ord) : Prop :=
  forall M, Included M B -> Inhabited _ M -> countable M ->
                            In B (|_| M).

Theorem AP_closed : Closed AP.

Theorem 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 phi0_elim : forall P : (Ord->Ord)->Prop,
    (forall f: Ord->Ord,
        ordering_function f ordinal AP -> P f) ->
    P phi0.
Proof.
  intros P H; apply H, phi0_ordering.
Qed.

Lemma AP_phi0 (alpha : Ord) : In AP (phi0 alpha).
Proof.
  pattern phi0; apply phi0_elim.
  destruct 1 as [H [H0 H1]];  apply H0;auto; split.
Qed.


Lemma phi0_zero : phi0 zero =  1.

Lemma phi0_mono (alpha beta : Ord) :
  alpha < beta ->  phi0 alpha < phi0 beta.

Lemma phi0_mono_weak (alpha beta : Ord) :
  alpha <= beta ->  phi0 alpha <= phi0 beta.

Lemma phi0_mono_R (alpha beta : Ord) :
  phi0 alpha < phi0 beta -> alpha < beta.
```

```
Lemma phi0_mono_R_weak : forall alpha beta,
    phi0 alpha <= phi0 beta -> alpha <= beta.

Lemma phi0_inj (alpha beta : Ord) :
  phi0 alpha = phi0 beta -> alpha = beta.


Lemma phi0_positive (alpha : Ord):  zero < phi0 alpha.


Lemma plus_lt_phi0 : forall ksi alpha,
    ksi < phi0 alpha ->
    ksi + phi0 alpha = phi0 alpha.


Lemma phi0_alpha_phi0_beta :
  forall alpha beta, alpha < beta ->
                     phi0 alpha + phi0 beta =
                     phi0 beta.


Lemma phi0_sup : forall U: Ensemble Ord,
    Inhabited _ U ->
    countable U ->
    phi0 (|_| U) = |_| (image U phi0).


Lemma phi0_of_limit (alpha : Ord)  :
  is_limit alpha ->
  phi0 alpha = |_| (image (members alpha) phi0).

Lemma AP_to_phi0 (alpha : Ord) :
  AP alpha -> exists beta,  alpha = phi0 beta.


Lemma AP_plus_AP (alpha beta gamma : Ord) :
  zero < beta ->
  phi0 alpha + beta = phi0 gamma ->
  alpha < gamma /\  beta = phi0 gamma.


Lemma is_limit_phi0 (alpha : Ord) :
  zero < alpha ->  is_limit (phi0 alpha).


Lemma omega_eqn : omega = phi0 1.

Lemma le_phi0 (alpha : Ord) : alpha <= phi0 alpha.
```

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.
```

Lets us denote by φ_α the ordering function of the set $Cr(\alpha)$ and by A_α its ordering segment.

```
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 zero = 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 a transfinite induction on α of the conjunction of both properties.

```
Theorem Unbounded_Cr alpha : Unbounded (Cr alpha).

Theorem Closed_Cr alpha : Closed (Cr alpha).
```

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].

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 $\leq$

[1] This would prevent us from trying to represent Cantor normal forms as finite trees (like in Sect. 4.1.2)

- 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'.

Theorem cnf_exists_unique (alpha:Ord) :
  exists! l: cnf_t, is_cnf_of alpha l.
Proof.
    destruct (cnf_exists alpha) as [l Hl]; exists l; split; auto.
    now apply cnf_unicity.
Qed.
```

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 ?

You may start this exercise with the file exercises/ordinals/schutte_cnf_counter_example.v.

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/oridinals/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 forms.

```
Theorem inject_of_zero : inject T1.zero = zero.
Proof. reflexivity. Qed.


Theorem inject_of_finite (n : nat):
  inject (T1.fin n) =  n.
```

```
Theorem inject_of_omega :
  inject T1.omega = Schutte_basics._omega.


Theorem inject_of_phi0 (alpha : T1):
  inject (T1.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.
```

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.

The Gaia project `https://github.com/coq-community/gaia` maintains Grimm's theory of ordinals as part of coq-community on GitHub. Integration of the present ordinal theory with Gaia, i.e., relating the different notions of ordinals and transferring relevant results, is an interesting project. First experiments in that direction are developped in the theories/gaia/ directory.

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`

```
Declare Scope T2_scope.
Delimit Scope T2_scope with t2.

Open Scope T2_scope.

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


Notation "[ alpha , beta ]" := (gcons alpha beta 0 zero)
                                 (at level 0): T2_scope.


Definition psi alpha beta  := [alpha, beta].

Definition psi_term alpha :=
  match alpha with zero => zero
                 | gcons a b n c => [a, b]
  end.
```

Like in chapter 4, we get familiar with the type T2 by recognizing simple constructs like finite ordinals, ω, etc., as inhabitants of T2.

```
Notation  one  := [zero,zero].
```

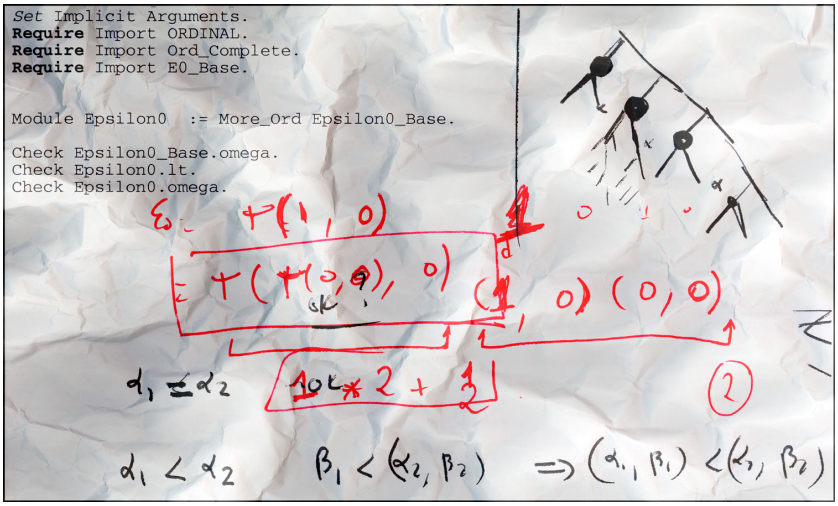


Figure 8.1: Veblen normal form

```
Notation FS n := (gcons zero zero n zero).

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

Coercion fin  : nat >-> T2.

Notation omega  := [zero,one].

Notation epsilon0  := ([one,zero]).

Definition epsilon alpha := [one, alpha].
```

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.
```

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`.

```
Example Ex1: 0 t2< epsilon0.
Proof.  constructor 1. Qed.

Example Ex2: omega t2< epsilon0.
Proof. info_auto with T2. (* uses lt_1 and lt_2 *) Qed.

Example Ex3: gcons omega 8 12 56 t2<  gcons omega 8 12 57.
Proof.
  constructor 7; constructor 6; auto with arith.
Qed.


Example Ex4: epsilon0 t2< [2,1].
Proof.
  apply lt_2; auto with T2.
  - apply lt_6; auto with arith.
Qed.
```

```
Example Ex5 : [2,1] t2< [2,3].
Proof.
  constructor 3; auto with T2.
  - constructor 6; auto with arith.
Qed.

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.


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 form

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')).

Global Hint Constructors nf : T2.
```

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.

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}.
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.

Compute compare (gcons 2 1 42 epsilon0) [2,2].
```

```
= Lt
: comparison
```

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 nfb  (gcons 2 1 42 epsilon0).
```

```
= true
: bool
```

```
Compute nfb (gcons 2 1 42 (gcons 2 2 4 epsilon0)).
```

```
= false
: bool
```

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. 4.1.8.2), the definition of addition in `T2` requires comparison between ordinal terms.
From *Gamma0.Gamma0*

```
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 153, 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.


Example Ex8:  phi 1 (succ epsilon0) = [1, [1,0] + 1].
Proof. reflexivity. Qed.
```

Despite its complexity, the function `phi` is well adapted to proofs by simplification or computation.

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 $\psi(\alpha, \gamma) = \phi_\alpha(\gamma + 1)$.

```
Lemma phi_psi : forall  beta gamma n,
    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.

```
Theorem phi_of_psi  : forall a b1 b2,
    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 4.1.4 on page 75).

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 50).

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. 4.1.6.1 on page 78).

```
Module G0.

  Definition LT := restrict nf lt.

Class G0 := mkg0 {vnf : T2; vnf_ok : nfb vnf}.

Definition lt (alpha beta : G0) := T2.lt (@vnf alpha) (@vnf beta).

Definition compare alpha beta := compare (@vnf alpha) (@vnf beta).
```

Then, we buils an instance of class `ON`. function `compare` is correct.

```
Instance lt_sto : StrictOrder lt.

Lemma lt_wf : well_founded lt.

Instance Gamma0_comp: Comparable lt compare.

Instance Gamma0: ON lt  compare.
```

Remark 8.1 The proof of `lt_wf` has been written by Évelyne Contejean, using her library on the recursive path ordering (see also remark 4.3 on page 86).

Project 8.6 Prove that `Epsilon0` (page 86) is a sub-notation system of `Gamma0`.

Prove that the implementations 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

Primitive recursive functions

9.1 Introduction

Primitive recursive functions are a small class of total functions, corresponding to the expressive power of a simple imperative programming language without **while** loops, in which every program execution terminates.

Primitive recursive functions are total functions from $\mathbb{N}^n$ to $\mathbb{N}$, for some $n \in \mathbb{N}$. Note that not all total n-ary recursive functions are primitive recursive (see for instance Sect. 9.5 on page 179).

The traditional definition of the set of primitive recursive functions is structured as an inductive definition in five rules: three base cases, and two recursive construction rules.

zero the constant function of value 0 is primitive recursive.

S The successor function $S : \mathbb{N} \to \mathbb{N}$ is primitive recursive.

projections For any pair $0 < i \leq n$, the projection $\pi_{i,n} : \mathbb{N}^n \to \mathbb{N}$, defined by $\pi_{i,n}(x_1, x_2, \ldots, x_n) = x_i$, is primitive recursive.

composition For any n and m, if $h : \mathbb{N}^m \to \mathbb{N}$, and $g_0, \ldots, g_{m-1} : \mathbb{N}^n \to \mathbb{N}$ are primitive recursive of n arguments, then the function which maps any tuple $(x_0, \ldots, x_{n-1})$ to $h(g_0(x0, \ldots, x_{n-1}), \ldots, g_{m-1}(x0, \ldots, x_{n-1})) : \mathbb{N}^n \to \mathbb{N}$ is primitive recursive.

primitive recursion If $g : \mathbb{N}^n \to \mathbb{N}$ and $h : \mathbb{N}^{n+2} \to \mathbb{N}$ are primitive recursive, then the function from $\mathbb{N}^{n+1}$ into $\mathbb{N}$ defined by

$$f(0, x_1, \ldots, x_n) = g(x_1, \ldots, x_n) \tag{9.1}$$

$$f(S(p), x_1, \ldots, x_n) = h(p, f(p, x_1, \ldots, x_n), x_1, \ldots, x_n) \tag{9.2}$$

is primitive recursive.

Please note the use of dots: ... in the definition above. Dots are not part of Gallina's syntax. Thus, the formal definition of the set of primitive recursive function will have to overcome this representation problem.

We present in this chapter a formalization of primitive recursive functions, taken from Russel O'Connor's formalization in Coq of Gödel's incompleteness theorems [O'C05].

Remark 9.1 The theory of primitive recursive function is now hosted in the `theories/ordinals/Ackermann` directory. The specific part on Gödel's theorem, is also on coq-community (`https://github.com/coq-community/goedel`) and requires the Pocklington library for lemmas on primality.

This chapter contains some comments on Russel's library, as well as a few extensions. Contributions (under the form of comments, new examples or exercises) are welcome!.

9.2 First look at the Ackermann library

O'Connor's library on Gödel's incompleteness theorems contains a little more than 45K lines of scripts. The part dedicated to primitive recursive functions and Peano arithmetic is 32K lines long and is originally structured in 38 modules. Thus, we propose a partial exploration of this library, through examples and exercises. Our additions to the original library — mainly examples and counter-examples —, are stored in the directory `theories/ordinals/MoreAck`.

In particular, the library MoreAck.AckNotPR contains the well-known proof that the Ackermann function is not primitive recursive (see Section 9.5 on page 179). Moreover, the library Hydra.Hydra_Theorems contains a proof that the length of an hydra battle (according to the initial replication factor) is not primitive recursive in general.

9.3 Basic definitions

The formal definition of primitive recursive functions lies in the library Ackermann.primRec, with preliminary definitions in Ackermann.extEqualNat and Ackermann.misc.

9.3.1 Functions of arbitrary arity

The `primRec` library allows us to consider primitive functions on `nat`, with any number of arguments, in curried form. This is made possible in Ackermann.extEqualNat by the following definition:

```
Fixpoint naryFunc (n : nat) : Set :=
  match n with
  | O => nat
  | S n => nat -> naryFunc n
  end.
```

For instance (`naryFunc 1`) is convertible to `nat -> nat` and (`naryFunc 3`) to `nat -> nat -> nat -> nat`.

From MoreAck.PrimRecExamples.

```
Require Import primRec.
Import extEqualNat.

Compute naryFunc 3.
```

```
= nat -> nat -> nat -> nat
: Set
```

```
Check plus: naryFunc 2.

Check 42: naryFunc 0.

Check (fun n p q : nat =>  n * p + q): naryFunc 3.
```

Likewise, arbitrary boolean predicates may have an arbitrary number of arguments. The dependent type (`naryRel` n), defined in the same way as `naryFunc`, is the type of n-ary functions from `nat` into `bool`.

```
Compute naryRel 2.
```

```
= nat -> nat -> bool
: Set
```

The magic of dependent types makes it possible to define recursively extensional equality between functions of the same arity.

From Ackermann.extEqualNat

```
Fixpoint  extEqual (n : nat) : forall  (a b : naryFunc n), Prop :=
  match n with
    0 => fun a b => a = b
  | S p => fun a b => forall c, extEqual p (a c) (b c)
  end.

Compute extEqual 2.
```

```
= fun a b : naryFunc 2 =>
  forall x x0 : nat, a x x0 = b x x0
: naryFunc 2 -> naryFunc 2 -> Prop
```

```
Example extEqual_ex1: extEqual 2 mult (fun x y =>  y * x + x - x).
Proof.
  intros x y.
```

```
x, y: nat
──────────────────────────────
extEqual 0 (x * y) (y * x + x - x)
```

```
  cbn.
```

```
x, y: nat
──────────────────────────────
x * y = y * x + x - x
```

Getting rid of the term `x-x`, we generate two easy-to-solve sub-goals.

```
  rewrite <- Nat.add_sub_assoc, Nat.sub_diag.
```

```
x, y: nat
──────────────────────────────
x * y = y * x + 0
```

```
x, y: nat
──────────────────────────────
x <= x
```

```
  - ring.
  - apply le_n.
Qed.
```

9.3.2 A Data-type for Primitive Recursive Functions

O'Connor's formalization of primitive recursive functions takes the form of two mutually inductive dependent data types, each constructor of which is associated with one of these rules. These two types are (`PrimRec` n) (primitive recursive functions of n arguments), and (`PrimRecs` n m) (m-tuples of primitive recursive functions of n arguments).

From Ackermann.primRec.

```
Inductive PrimRec : nat -> Set :=
  | succFunc : PrimRec 1
  | zeroFunc : PrimRec 0
  | projFunc : forall n m : nat, m < n -> PrimRec n
  | composeFunc :
      forall (n m : nat) (g : PrimRecs n m) (h : PrimRec m),
        PrimRec n
  | primRecFunc :
      forall (n : nat) (g : PrimRec n) (h : PrimRec (S (S n))),
        PrimRec (S n)
with PrimRecs : nat -> nat -> Set :=
  | PRnil : forall n : nat, PrimRecs n 0
  | PRcons : forall n m : nat, PrimRec n -> PrimRecs n m ->
                               PrimRecs n (S m).
```

Remark 9.2 Beware of the conventions used in the `primRec` library! The constructor (`projFunc` n m) is associated with the projection $\pi_{n-m,n}$ and *not* $\pi_{n,m}$. For instance, the projection $\pi_{2,5}$ defined by $\pi_{2,5}(a,b,c,d,e) = b$ corresponds to the term (`projFunc 5 3 H`), where `H` is a proof of $3 < 5$. This fact is reported in the comments of `primRec.v`. We presume that this convention makes it easier to define the evaluation function (`evalProjFunc` n) (see the next sub-section). Trying the other convention is left as an exercise.

9.3.3 A little bit of semantics

Please note that inhabitants of type (`PrimRec` n) are not Coq functions like `Nat.mul`, or factorial, etc. The data-type (`PrimRec` n) is indeed an abstract syntax for the language of primitive recursive functions. The bridge between

this language and the word of usual functions is an interpretation function (`evalprimRec` n) of type `PrimRec` $n \to$ `naryFunc` n. This function is defined by mutual recursion, together with the function (`evalprimRecS` n m) of type `PrimRecs` $n\,m \to$ `Vector.t` (`naryFunc` n) m.

Both functions are mutually defined through dependent pattern matching. We advise the readers who would feel uneasy with dependent types to consult Adam Chlipala's *cpdt* book [Chl11]. We leave it to the reader to look also at the helper functions in Ackermann.primRec.

```
Fixpoint evalPrimRec (n : nat) (f : PrimRec n) {struct f} :
 naryFunc n :=
  match f in (PrimRec n) return (naryFunc n) with
  | succFunc => S
  | zeroFunc => 0
  | projFunc n m pf => evalProjFunc n m pf
  | composeFunc n m l f =>
      evalComposeFunc n m (evalPrimRecs _ _ l) (evalPrimRec _ f)
  | primRecFunc n g h =>
      evalPrimRecFunc n (evalPrimRec _ g) (evalPrimRec _ h)
  end

 with evalPrimRecs (n m : nat) (fs : PrimRecs n m) {struct fs} :
 Vector.t (naryFunc n) m :=
  match fs in (PrimRecs n m) return (Vector.t (naryFunc n) m) with
  | PRnil a => Vector.nil  (naryFunc a)
  | PRcons a b g gs =>
    Vector.cons _ (evalPrimRec _ g) _ (evalPrimRecs _ _ gs)
  end.
```

Looks complicated? The following examples show that, when the arity is fixed, these definitions behave well w.r.t. Coq's reduction rules. Moreover, they make the interpretation functions more "concrete".

From MoreAck.PrimRecExamples.

```
Example Ex1 : evalPrimRec 0 zeroFunc = 0.
Proof. reflexivity. Qed.

Example Ex2 a : evalPrimRec 1 succFunc a = S a.
Proof. reflexivity. Qed.

Example Ex3 a b c d e f: forall (H: 2 < 6),
    evalPrimRec 6
                (projFunc 6 2 H) a b c d e f = d.
Proof. reflexivity. Qed.


Example Ex4 (x y z : PrimRec 2) (t: PrimRec 3):
  let u := composeFunc 2 3
                       (PRcons 2 _ x
                               (PRcons 2 _ y
```

```
                                        (PRcons 2 _ z
                                                (PRnil 2))))
                        t in
  let f := evalPrimRec 2 x in
  let g := evalPrimRec 2 y in
  let h := evalPrimRec 2 z in
  let i := evalPrimRec 3 t in
  let j := evalPrimRec 2 u in
  forall a b, j a b = i (f a b) (g a b) (h a b).
Proof. reflexivity. Qed.

Example Ex5 (x : PrimRec 2)(y: PrimRec 4):
  let g := evalPrimRec _ x in
  let h := evalPrimRec _ y in
  let f := evalPrimRec _ (primRecFunc _ x y) in
  forall a b,  f 0 a b = g a b.
Proof. reflexivity.   Qed.

Example Ex6 (x : PrimRec 2)(y: PrimRec 4):
  let g := evalPrimRec _ x in
  let h := evalPrimRec _ y in
  let f := evalPrimRec _ (primRecFunc _ x y) in
  forall n a b, f (S n) a b = h n (f n a b) a b.
Proof. reflexivity.   Qed.
```

Another example? Let us consider the following term[1]:

```
Example bigPR : PrimRec 1 :=
primRecFunc 0
  (composeFunc 0 1 (PRcons 0 0 zeroFunc (PRnil 0)) succFunc)
  (composeFunc 2 2
    (PRcons 2 1
      (composeFunc 2 1
         (PRcons 2 0 (projFunc 2 1 (le_n 2))
                 (PRnil 2))
         succFunc)
      (PRcons 2 0
        (composeFunc 2 1
          (PRcons 2 0
             (projFunc 2 0
                       (le_S 1 1 (le_n 1)))
             (PRnil 2))
          (projFunc 1 0 (le_n 1))) (PRnil 2)))
    (primRecFunc 1 (composeFunc 1 0 (PRnil 1) zeroFunc)
       (composeFunc 3 2
         (PRcons 3 1
            (projFunc 3 1 (le_S 2 2 (le_n 2)))
            (PRcons 3 0 (projFunc 3 0
```

[1]Of course, we never typed this term *verbatim*; we obtained it by an interactive proof the reader will be able to make after reading Sect.9.4 on the facing page.

```
                    (le_S 1 2
                          (le_S 1 1 (le_n 1))))
              (PRnil 3)))
      (primRecFunc 1 (projFunc 1 0 (le_n 1))
                  (composeFunc 3 1
                      (PRcons 3 0
                              (projFunc 3 1 (le_S 2 2 (le_n 2)))
                              (PRnil 3))
                      succFunc))))).
```

Let us now interpret this term as an arithmetic function.

```
Example  mystery_fun : nat -> nat := evalPrimRec 1 bigPR.

Compute map mystery_fun [0;1;2;3;4;5;6] : t nat _.
```

```
= [1; 1; 2; 6; 24; 120; 720]
: t nat 7
```

After this test, the term `bigPR` looks to be a primitive recursive definition of the factorial function, although we haven't proved this fact yet. Fortunately, we will see in the following sections simple ways to prove that a given function is primitive recursive, without building such an unreadable term.

9.4 Proving that a given arithmetic function is primitive recursive

The example in the preceding section clearly shows that, in order to prove that a given arithmetic function (defined in Gallina as usual) is primitive recursive, trying to give by hand a term of type (`PrimRec` n) is not a good method, since such terms may be huge and complex, even for simple arithmetic functions. The method proposed in Library `primRec` is the following one:

1. Define a type corresponding to the statement "the function f:`naryFunc` n is primitive recursive ".
2. Prove handy lemmas which may help to prove that a given function is primitive recursive.

Thus, the proof that a function, like `factorial`, is primitive recursive may be interactive, without having to type complex terms at any step of the development.

9.4.1 The predicate `isPR`

Let f be an arithmetic function of arity n. We say that f is primitive recursive if f is **extensionally** equal to the interpretation of some term of type `PrimRec` n.

From Ackermann.primRec.

```
Definition isPR (n : nat) (f : naryFunc n) : Set :=
  {p : PrimRec n | extEqual n (evalPrimRec _ p) f}.
```

The library `primRec` contains a large catalogue of lemmas allowing to prove statements of the form (`isPR` n f). We won't list all these lemmas here, but give a few examples of how they may be applied.

Remark 9.3 In the library `primRec`, all these lemmas are opaque (registered with `Qed`. Thus they do not allow the user to look at the witness of a proof of a `isPR` statement. Our example of page172 was built using a copy of `primRec.v` where many `Qed`s have been replaced with `Defined`s.

If it does not cause compatibility problems (with goedel library for instance), we plan to make all theses lemmas transparent.

9.4.1.1 Elementary proofs of `isPR` statements

The constructors `zeroFunc`, `succFunc`, and `projFunc` of type `PrimRec` allows us to write trivial proofs of primitive recursivity. Although the following lemmas are already proven in Ackermann.primRec, we wrote alternate proofs in Ackermann.MoreAck.PrimRecExamples.v, in order to illustrate the main proof patterns.

```
Lemma zeroIsPR : isPR 0 0.
Proof.
  exists zeroFunc.
```

```
extEqual 0 (evalPrimRec 0 zeroFunc) 0
```

```
  cbn.
```

```
0 = 0
```

```
  reflexivity.
Qed.
```

Likewise, we prove that the successor function on `nat` is primitive recursive too.

```
Lemma SuccIsPR : isPR 1 S.
Proof.
  exists succFunc; cbn; reflexivity.
Qed.
```

Projections are proved primitive recursive, case by case (many examples in Ackermann.primRec). *Please notice again that the name of the projection follows the mathematical tradition, whilst the arguments of* ***`projFunc`*** *use another convention (*cf *remark 9.2 on page 170).*

```
Lemma pi2_5IsPR : isPR 5 (fun a b c d e => b).
Proof.
 assert (H: 3 < 5) by auto.
 exists (projFunc 5 3 H).
 cbn; reflexivity.
Qed.
```

Please note that the projection $\pi_{1,1}$ is just the identity on `nat`, and is realized by (`projFunc 1 0`).

From Ackermann.primRec.

```
Lemma idIsPR : isPR 1 (fun x : nat => x).
Proof.
  assert (H: 0 < 1) by auto.
  exists (projFunc 1 0 H); cbn; auto.
Qed.
```

9.4.1.2 Using function composition

Let us look at the proof that any constant n of type `nat` has type (`PR 0`) (lemma `const1_NIsPR` of `primRec`). We carry out a proof by induction on n, the base case of which is already proven. Now, let us assume n is `PR n`, with x : `PrimRec 0` as a "realizer". Thus we would like to compose this constant function with the unary successor function.

This is exactly the role of the instance `composeFunc 0 1` of the dependently typed function `composeFunc`, as shown by the following lemma.

```
Fact compose_01 :
    forall (x:PrimRec 0) (t : PrimRec 1),
    let c := evalPrimRec 0 x in
    let f := evalPrimRec 1 t in
    evalPrimRec 0 (composeFunc 0 1
                               (PRcons 0 0 x (PRnil 0))
                               t)  =
     f c.
Proof. reflexivity. Qed.
```

Thus, we get a quite simple proof of `const1_NIsPR`.

From MoreAck.PrimRecExamples.

```
Lemma  const0_NIsPR n : isPR 0 n.
Proof.
  induction n.
 - apply zeroIsPR.
 - destruct IHn as [x Hx].
   exists (composeFunc 0 1 (PRcons 0 0 x (PRnil 0)) succFunc).
   cbn in *; intros; now rewrite Hx.
Qed.
```

9.4.1.3 Proving that `plus` is primitive recursive

The lemma `plusIsPR` is already proven in Ackermann.primRec. We present in MoreAck.PrimRecExamples a commented version of this proof,

First, we look for lemmas which may help to prove that a given function obtained with the recursor `nat_rec` is primitive recursive.

```
Search (isPR 2 (fun _ _ => nat_rec _ _ _ _)).
```

```
ind1ParamIsPR:
  forall f : nat -> nat -> nat -> nat,
  isPR 3 f ->
  forall g : nat -> nat,
  isPR 1 g ->
  isPR 2
    (fun a b : nat =>
     nat_rec (fun _ : nat => nat) (g b)
       (fun x y : nat => f x y b) a)
```

The following lemma shows it suffices to prove that Standard library's function `plus` is extensionally equal to a function defined with `nat_rec`.

```
Lemma isPR_extEqual_trans n f g :
  isPR n f -> extEqual n f g -> isPR n g.
Proof.
 intros [x Hx]; exists x.
 apply extEqualTrans with f; auto.
Qed.
```

Thus, let us define an helper and prove its equivalence with `plus`.

```
Definition plus_alt x y  :=
              nat_rec  (fun n : nat => nat)
                       y
                       (fun z t =>  S t)
                       x.

Lemma plus_alt_ok:
  extEqual 2 plus_alt plus.
Proof.
  intro x; induction x; cbn; auto.
  intros y; cbn; now rewrite <- (IHx y).
Qed.
```

We are now able to complete the proof.

```
Lemma plusIsPR : isPR 2 plus.
Proof.
  apply isPR_extEqual_trans with plus_alt.
```

```
______________________________________
isPR 2 plus_alt
______________________________________
extEqual 2 plus_alt Init.Nat.add
```

```
  -  unfold plus_alt; apply ind1ParamIsPR.
```

```
______________________________________
isPR 3 (fun _ y _ : nat => S y)
______________________________________
isPR 1 (fun b : nat => b)
```

We already proved that `S` is `PR 1`, but we need to consider a function of three arguments, which ignores its first and third arguments. Fortunately, the library `primRec` already contains lemmas adapted to this kind of situation.

```
filter010IsPR
     : forall g : nat -> nat,
       isPR 1 g -> isPR 3 (fun _ b _ : nat => g b)
```

Thus, our first subgoal is solved easily. The rest of the proof is just an application of already proven lemmas.

```
    + apply filter010IsPR, succIsPR.
    + apply idIsPR.
  - apply plus_alt_ok.
Qed.
```

To do 9.1 *Comment more examples from MoreAck.PrimRecExamples.*

Exercise 9.1 There is a lot of lemmas similar to `filter010IsPR` in the `primRec` library, useful to control the arity of functions. Thus, the reader may look at them, and invent simple examples of application for each lemma.

Exercise 9.2 Multiplication of natural number is already proven in the `primRec` library. Write a proof of your own, then compare to the library's version.

9.4.1.4 More examples

The following proof decomposes the `double` function as the composition of multiplication with the identity and the constant function which returns 2. *Note that the lemma `const1_NIsPR` considers this function as an unary function (unlike `const0_NIsPR`).*

```
Definition double (n:nat) := 2 * n.

Lemma doubleIsPR : isPR 1 double.
Proof.
  unfold double; apply compose1_2IsPR.
```

```
3 subgoals (ID 184)

  ============================
  isPR 1 (fun _ : nat => 2)

subgoal 2 (ID 185) is:
 isPR 1 (fun x : nat => x)
subgoal 3 (ID 186) is:
 isPR 2 Init.Nat.mul
```

```
  - apply const1_NIsPR.
  - apply idIsPR.
  - apply multIsPR.
Qed.
```

Exercise 9.3 Prove that the following functions are primitive recursive.

```
Fixpoint fact n :=
  match n with 0 => 1
          |   S p  => n * fact p
  end.

Fixpoint exp n p :=
  match p with
    0 => 1
  | S m =>  exp n m * n
  end.

Fixpoint tower2 n :=
  match n with
    0 => 1
  | S p => exp 2 (tower2 p)
  end.
```

Hint: You may have to look again at the lemmas of the library Ackermann.primRec if you meet some difficulty. You may start this exercise with the file exercises/primrec/MorePRExamples.v.

Exercise 9.4 Show that the function `min: naryFunc 2` is primitive recursive.
You may start this exercise with exercises/primrec/MinPR.v.

Exercise 9.5 Write a simple and readable proof that the Fibonacci function is primitive recursive.

```
Fixpoint fib (n:nat) : nat :=
  match n with
  | 0 => 1
  | 1 => 1
  | S ((S p) as q) => fib q + fib p
  end.
```

Hint: You may use as a helper the function which computes the pair $(\texttt{fib}(n+1), \texttt{fib}(n))$. Library Ackermann.cPair contains the definition of the encoding of $\mathbb{N}^2$ into $\mathbb{N}$, and the proofs that the associated constructor and projections are primitive recursive. *You may start this exercise with the file exercises/primrec/FibonacciPR.v.*

Exercise 9.6 Prove the following lemmas (which may help to solve the next exercise).

```
Lemma boundedSearch3 :
  forall (P : naryRel 2) (b  : nat), boundedSearch P b <= b.

Lemma boundedSearch4 :
  forall (P : naryRel 2) (b  : nat),
    P b b = true ->
    P b (boundedSearch P b) = true.
```

Note: The function `boundedSearch` is defined in Library Ackermann.primRec.

Exercise 9.7 Prove that the function which returns the integer square root of any natural number is primitive recursive.

You may start this exercise with the file exercises/primrec/isqrt.v.

9.5 Proving that a given function is *not* primitive recursive

The best known example of a total recursive function which is not primitive recursive is the Ackermann function. We show how to adapt the classic proof (see for instance [Pla13]) to the constraints of Gallina. We hope this formal proof is a nice opportunity to explore the treatment of primitive recursive functions by R. O'Connor, and to play with dependent types.

9.5.1 Ackermann function

Ackermann function is traditionally defined as a function from $\mathbb{N} \times \mathbb{N}$ into $\mathbb{N}$, through three equations:

$$A(0, n) = n + 1 \tag{9.3}$$
$$A(m + 1, 0) = A(m, 1) \tag{9.4}$$
$$A(m + 1, n + 1) = A(m, A(m + 1, n)) \tag{9.5}$$

Let us try to define this function in Coq (in curried form).

```
Fail
  Fixpoint Ack (m n : nat) : nat :=
  match m, n with
  | 0, n => S n
  | S m, 0 => Ack m 1
  | S m0, S p => Ack m0 (Ack m p)
  end.
```

```
The command has indeed failed with message:
Cannot guess decreasing argument of fix.
```

A possible workaround is to make `m` be the decreasing argument, and define — within `m`'s scope — a local helper function which computes (`Ack m n`) for any `n`. This way, both functions `Ack` and `Ackm` have a (structurally) strictly decreasing argument.

```
Module Alt.

  Fixpoint Ack (m n : nat) : nat :=
    match m with
    | O => S n
    | S p => let fix Ackm (n : nat) :=
                 match n with
                 | O => Ack p 1
                 | S q => Ack p (Ackm q)
                 end
             in Ackm n
    end.

  Compute Ack 3 2.
```

```
= 29
: nat
```

```
End Alt.
```

We preferred to define a variant which uses explicitly the functional `iterate`, where (`iterate` f n) is the n-th iteration of f [2]. It makes it possible to apply a few lemmas proved in Prelude.Iterates, for instance about the monotony of the n-th iterate of a given function.

From 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.

Lemma iterate_le_n_Sn (f: nat -> nat):
  (forall x,  x <= f x) ->
  forall n x,  iterate f n x <= iterate f (S n)  x.
```

Thus, our definition of the Ackermann function is as follows:

From MoreAck.Ack.

```
Fixpoint Ack (m:nat) : nat -> nat :=
  match m with
  | 0 => S
  | S n => fun k =>  iterate (Ack n) (S k) 1
  end.

Compute Ack 3 2.
```

```
= 29
: nat
```

[2] Please not confuse with `primRec.iterate`, which is monomorphic and does not share the same order of arguments.

Exercise 9.8 The file MoreAck.Ack presents two other definitions of the Ackermann functions based on the lexicographic ordering on $\mathbb{N} \times \mathbb{N}$. Prove that the four functions are extensionally equal.

9.5.1.1 First properties of the Ackermann function

The three first lemmas make us sure that our function `Ack` satisfies the "usual" equations.

```
Lemma Ack_0 : Ack 0 = S.
Proof refl_equal.

Lemma Ack_S_0 m : Ack (S m) 0 = Ack m 1.
Proof. reflexivity. Qed.

Lemma Ack_S_S : forall m p,
    Ack (S m) (S p) = Ack m (Ack (S m) p).
Proof. reflexivity. Qed.
```

The order of growth of the Ackermann function w.r.t. its first argument is illustrated by the following equalities.

```
Lemma Ack_1_n n : Ack 1 n = S (S n).

Lemma Ack_2_n n: Ack 2  n = 2 * n + 3.

Lemma Ack_3_n n: Ack 3 n = exp2 (S (S (S n))) - 3.

Lemma Ack_4_n n : Ack 4 n = hyper_exp2 (S (S (S n))) - 3.
```

Remark 9.4 The statements above can be rewritten in a more uniform way:

For $m \in 1..4$, $\texttt{Ack}\, m\, n = f_m\,(n+3) - 3$, where

$$\begin{aligned} f_1(n) &= n + 2 \\ f_2(n) &= n \times 2 \\ f_3(n) &= 2^n \\ f_4(n) &= 2^{2^{\cdots^2}} \quad (n \textit{ levels}) \end{aligned}$$

An important property of the Ackermann function helps us to overcome the difficulty raised by nested recursion, by climbing up the hierarchy $\texttt{Ack}\, n\, \_$ ($n \in \mathbb{N}$).
From MoreAck.Ack.

```
Lemma nested_Ack_bound k m n :
  Ack k (Ack m n) <= Ack (2 + max k m) n.
```

Please note also that for any given n, the unary function $(\texttt{Ack}\, n)$ is primitive recursive.

From MoreAck.AckNotPR.

```
Theorem Ackn_IsPR (n: nat) : isPR 1 (Ack n).
Proof.
  induction n.
```

9.5.2 A proof by induction on all primitive recursive functions

In order to prove that `Ack` (considered as a function of two arguments) is not primitive recursive, the usual method consists in two steps:

1. Prove that for any primitive recursive function $f : \mathbb{N} \to \mathbb{N} \to \mathbb{N}$, there exists some natural number n depending on f, such that, for any x and y, $f\,x\,y \leq \texttt{Ack}\,n\,(\max x\,y)$ (we say that f is *"majorized"* by `Ack`).

2. Show that `Ack` fails to satisfy this property.

First, we prove that any primitive function of two arguments is majorized by `Ack`. If we look at the inductive definition of primitive recursive functions, page 170, it is obvious that a proof by induction on the construction of primitive recursive functions must consider functions of any arity.

The following scheme allows us to write proofs by induction on the class of primitive recursive functions.

From Ackermann.primRec.

```
Scheme PrimRec_PrimRecs_ind := Induction for PrimRec Sort Prop
  with PrimRecs_PrimRec_ind := Induction for PrimRecs  Sort Prop.

Check PrimRec_PrimRecs_ind.
```

```
PrimRec_PrimRecs_ind
     : forall (P : forall n : nat, PrimRec n -> Prop)
         (P0 : forall n n0 : nat,
               PrimRecs n n0 -> Prop),
       P 1 succFunc ->
       P 0 zeroFunc ->
       (forall (n m : nat) (l : m < n),
        P n (projFunc n m l)) ->
       (forall (n m : nat) (g : PrimRecs n m),
        P0 n m g ->
        forall h : PrimRec m,
        P m h -> P n (composeFunc n m g h)) ->
       (forall (n : nat) (g : PrimRec n),
        P n g ->
        forall h : PrimRec (S (S n)),
        P (S (S n)) h -> P (S n) (primRecFunc n g h)) ->
       (forall n : nat, P0 n 0 (PRnil n)) ->
       (forall (n m : nat) (p : PrimRec n),
        P n p ->
        forall p0 : PrimRecs n m,
        P0 n m p0 -> P0 n (S m) (PRcons n m p p0)) ->
       forall (n : nat) (p : PrimRec n), P n p
```

Please note that, in order to prove a property shared by any primitive recursive function of, say, arity 2, this induction scheme leads you to consider an extension of the considered property to primitive recursive function of any arity.

Thus the lemma we will have to prove is the following one:

> For any n, and any primitive recursive function f of arity n, there exists some natural number q such that the following inequality holds:
>
> $$\forall x_1, \ldots, x_n, f(x_1, \ldots, x_n) \leq \mathrm{Ack}(q, \max(x_1, \ldots, x_n))$$

But dots don't belong to Gallina's syntax! So, we may use Coq's vectors for denoting arbitrary tuples.

First, we extend `max` to vectors of natural numbers (using the notations of module `VectorNotations` and some more definitions from Prelude.MoreVectors). So, (`t` A n) is the type of vectors of n elements of type A, and the constants `cons`, `nil`, `map`, etc., refer to vectors and not to lists. Likewise, the notation `x::v` is an abbreviation for `VectorDef.cons x _ v`.

```
Fixpoint max_v {n:nat} (v: Vector.t nat n) : nat :=
  match v  with
  | nil =>  0
  | cons x t => max x (max_v t)
  end.

Lemma max_v_2 : forall x y,  max_v (x::y::nil) = max x y.

Lemma max_v_lub : forall n (v: t nat n) y,
    (Forall (fun x =>  x <= y) v) ->
    max_v v <= y.

Lemma max_v_ge : forall n (v: t nat n) y,
    In  y  v -> y <= max_v v.
```

We have also to convert any application ($f\, x_1\, x_2\, \ldots\, x_n$) into an application of a function to a single argument: the vector of all the x_is. This is already defined in Library Ackermann.primRec.

```
Fixpoint evalList (m : nat) (l : Vector.t nat m) {struct l} :
  naryFunc m -> nat :=
  match l  in (Vector.t _ m) return (naryFunc m -> nat) with
  | Vector.nil => fun x : naryFunc 0 => x
  | Vector.cons a n l' =>
    fun x : naryFunc (S n) => evalList n l' (x a)
  end.
```

Indeed, (`evalList` m v f) is the application to the vector v of an uncurried version of f. In LibraryMoreAck.AckNotPR, we introduce a lighter notation.

```
Notation "'v_apply' f v" := (evalList _ v f) (at level 10, f at level 9).


Example Ex2 : forall (f: naryFunc 2) x y,
    v_apply f (x::y::nil) = f x y.
Proof.
  intros; now cbn.
Qed.
```

```
Example Ex4 : forall (f: naryFunc 4) x y z t,
    v_apply f (x::y::z::t::nil) = f x y z t.
Proof.
  intros; now cbn.
Qed.
```

We are now able to translate in Gallina the notion of "majorization":

```
(** ** Comparing an n-ary and a binary functions *)

Definition majorized {n} (f: naryFunc n) (A: naryFunc 2) :=
  exists (q:nat),
    forall (v: t nat n), v_apply f v <= A q (max_v v).

Definition majorizedPR {n} (x: PrimRec n) A :=
  majorized (evalPrimRec n x) A.

(** For vectors of functions *)

Definition majorizedS {n m} (fs : Vector.t (naryFunc n) m)
           (A : naryFunc 2):=
  exists N, forall (v: t nat n),
      max_v (map (fun f => v_apply f v) fs) <= A N (max_v v).

Definition majorizedSPR {n m} (x : PrimRecs n m) :=
  majorizedS (evalPrimRecs _ _ x).
```

Now, it remains to prove that any primitive function is majorized by `Ack`. The three base cases are as follows:

```
Lemma majorSucc : majorizedPR  succFunc Ack.

Lemma majorZero : majorizedPR  zeroFunc Ack.

Lemma majorProjection (n m:nat)(H: m < n):
  majorizedPR (projFunc n m H) Ack.
```

The remaining cases are proved within a mutual induction.

```
Lemma majorAnyPR: forall n (x: PrimRec n), majorizedPR  x Ack.
Proof.
  intros n x; induction x using PrimRec_PrimRecs_ind with
                  (P0 := fun n m y => majorizedSPR  y Ack).
  - apply majorSucc.
  - apply majorZero.
  - apply majorProjection.

  - destruct IHx, IHx0; red; exists (2 + max x0 x1).
```

```
x : PrimRec m
x0 : nat
```

```
H : forall v : t nat n,
  max_v (map (fun f : naryFunc n => v_apply f v)
          (evalPrimRecs n m g)) <=
  Ack x0 (max_v v)
x1 : nat
H0 : forall v : t nat m, v_apply (evalPrimRec m x) v <=
                                  Ack x1 (max_v v)
 ============================
 forall v : t nat n,
 v_apply (evalPrimRec n (composeFunc n m g x)) v <=
 Ack (2 + Nat.max x0 x1) (max_v v)
```

```
- destruct IHx1 as [r Hg]; destruct IHx2 as [s Hh].
```

```
 n : nat
 x1 : PrimRec n
 x2 : PrimRec (S (S n))
 r : nat
 Hg : forall v : t nat n,
    v_apply (evalPrimRec n x1) v <= Ack r (max_v v)
 s : nat
 Hh : forall v : t nat (S (S n)),
      v_apply (evalPrimRec (S (S n)) x2) v <= Ack s (max_v v)
 ============================
 majorizedPR (primRecFunc n x1 x2) Ack
```

The las two goals deal with vectors of functions.

```
1 subgoal (ID 289)

 n : nat
 ============================
 majorizedSPR (PRnil n) Ack
```

```
1 subgoal (ID 296)

 n, m : nat
 x : PrimRec n
 p : PrimRecs n m
 IHx : majorizedPR x Ack
 IHx0 : majorizedSPR p Ack
 ============================
 majorizedSPR (PRcons n m x p) Ack
```

```
Qed.
```

9.5.3 Looking for a contradiction

The following lemma is just a specialization of `majorAnyPR` to binary functions (forgetting vectors, coming back to usual notations).

```
Lemma majorPR2 (f: naryFunc 2)(Hf : isPR 2 f)
  : exists (n:nat), forall x y,  f x y <= Ack n (max x  y).
```

We prove also a strict version of this lemma, thanks to the following property (proved in Library MoreAck.Ack).

```
Lemma Ack_strict_mono_l : forall n m p, n < m ->
                                        Ack n (S p) < Ack m (S p).
```

From MoreAck.AckNotPR.

```
Lemma majorPR2_strict (f: naryFunc 2)(Hf : isPR 2 f):
    exists n:nat,
      forall x y, 2 <= x -> 2 <= y -> f x y < Ack n (max x y).
```

If the Ackermann function were primitive recursive, then there would exist some natural number n, such that, for all x and y, the inequality $\texttt{Ack}\,x\,y \leq \texttt{Ack}\,n\,(\texttt{max}\,x\,y)$ holds. Thus, our impossibility proof is just a sequence of easy small steps.

```
Section Impossibility_Proof.

  Hypothesis HAck : isPR 2 Ack.

  Lemma Ack_not_PR : False.
  Proof.
    destruct (majorPR2_strict Ack HAck) as [m Hm].
    pose (X := max 2 m); specialize (Hm X X).
    rewrite max_idempotent in Hm;
      assert (H0: Ack m X <= Ack X X) by (apply Ack_mono_l; lia).
    lia.
  Qed.


End Impossibility_Proof.
```

Remark 9.5 It is easy to prove that any unary function which dominates `fun n => Ack n n` fails to be primitive recursive. To this end, we use an instance of `majorAnyPR` dealing with unary functions.

From MoreAck.AckNotPR.

```
Lemma majorPR1  (f: naryFunc 1)(Hf : isPR 1 f)
  : exists (n:nat), forall x,  f x  <= Ack n x.
```

Then, we write a short proof by contradiction.

```
Section dom_AckNotPR.

  Variable f : nat -> nat.
  Hypothesis Hf : dominates f (fun n => Ack n n).
```

```
  Lemma dom_AckNotPR: isPR 1 f -> False.
  Proof.
    intros H;  destruct Hf as [N HN].
    destruct  (majorPR1 _ H) as [M HM].
    pose (X := Max.max N M).
    specialize (HN X  (Max.le_max_l N M)); (* for 8.13.dev's lia *)
      cbn in HN.
    specialize (HM X).
      assert (Ack M X <= Ack X X) by (apply Ack_mono_l; subst; lia).
    lia.
  Qed.

End dom_AckNotPR.
```

Remark 9.6 Note that the Ackermann function is a counter-example to the (false) statement:

> "Let f be a function of type `naryFunc 2`. If, for any n, the function $f(n)$ is primitive recursive, then f is primitive recursive."

9.6 The length of standard hydra battles

The module Hydra_Theorems contains a proof that the function which computes the length of standard hydra battles is not primitive recursive. More precisely, we consider, for a given hydra $h = \iota(\alpha)$, the length of a standard battle which starts with the replication factor k (see Sect 6.2.4.2 on page 123).

This proof is a little more complex than the preceding one.

9.6.1 Definitions

The function we consider is defined and proven correct in Module Hydra.Battle_length.

```
Definition l_std alpha k := (L_ alpha (S k) - k)%nat.

Lemma l_std_ok : forall alpha : E0,
    alpha <> Zero ->
    forall k : nat,
      1 <= k -> battle_length standard k (iota (cnf alpha))
                              (l_std alpha k).
```

9.6.2 Proof steps

Now, let us assume that the function `l_std` is primitive recursive.

From Hydra.Hydra_Theorems.

```
Section battle_lenght_notPR.

  Hypothesis H: forall alpha, isPR 1 (l_std alpha).
```

Let us consider the hydra represented by the ordinal ω^ω.

```
Let alpha := phi0 omega%e0.
Let h := iota (cnf alpha).
```

In order to get rid of the subtraction in the definition of `l_std`, we work with a helper function.

```
Let m k := L_ alpha (S k).

Remark m_eqn : forall k, m k = (l_std alpha k + k)%nat.
```

Under the hypothesis `H`, m is also primitive recursive.

```
Remark mIsPR : isPR 1 m.
```

9.6.2.1 Comparison between F and H'

In Epsilon0.F_alpha, we prove a relation between the F and H' functionals. For any α and $k > 0$, $H'_{\omega^\alpha}(k) \geq F_\alpha(k)$.

```
Lemma H'_F alpha : forall n,  F_ alpha (S n) <= H'_ (phi0 alpha) (S n).
Proof.
  pattern alpha; apply well_founded_induction with Lt.
```

Our proof of this lemma is not trivial at all, it uses some properties of the Ketonen-Solovay's toolkit. We advise the reader to explore this proof, with the help of an IDE or software like Alectryon.

9.6.2.2 End of the proof

We finish the proof by comparing several fast growing functions.

From Epsilon0.L_alpha

```
Theorem H'_L_ alpha :
  forall i:nat,  (H'_ alpha i <= L_ alpha (S i))%nat.
```

From Epsilon0.F_omega

```
Lemma F_vs_Ack n : 2 <= n -> Ack n n <= F_ omega n.
```

By transitivity, we get the inequality $F_\omega(k+1) \leq m(k+1)$, for any k.

```
Remark m_ge_F_omega : forall k,  F_ omega (S k) <= m (S k).
```

We finish the proof by noting that the function m (composed with `S`) dominates the Ackermann function, which leads to a contradiction.

```
Remark m_dominates_Ack :
  dominates (fun n =>  S (m n)) (fun n => Ack.Ack n n).

Remark SmNotPR : isPR 1 (fun n => S (m n)) -> False.
```

```
  Theorem LNotPR : False.
  Proof.
    apply SmNotPR,  compose1_1IsPR.
    - apply mIsPR.
    - apply succIsPR.
  Qed.

End battle_lenght_notPR.
```

Part II

A few certified algorithms

Chapter 10

Smart computation of x^n

10.1 Introduction

Nothing looks simpler than writing a function for computing x^n. But on the contrary, this simple programming exercise allows us to address advanced programming techniques such as:

- monadic programming, and continuation passing style
- type classes, and generalized rewriting
- proof engineering, in particular proof reuse
- proof by reflection
- polymorphism and parametricity
- composition of correct programs, etc.

10.2 Some basic implementations

Let us start with a very naive way of computing the n-th power of x, where n is a natural number and x belongs to some type for which a multiplication and an identity element are defined.

From Module ***additions.FirstSteps***

```
Section Definitions.

Variables (A: Type)
          (mult: A -> A -> A)
          (one: A).
Local Infix "*" := mult.
Local Notation "1" := one.
```

```
Fixpoint power (x:A)(n:nat) : A :=
  match n with
   | 0%nat => 1
   | S p =>   x * x ^ p
```

```
  end
where "x ^ n" := (power x n).
```

```
Compute power Z.mul 1%Z 2%Z 10.
```

```
  = 1024%Z
    : Z
```

```
Open Scope string_scope.
Compute power append  "" "ab"  12.
```

```
 = "abababababababababababab"
     : string
```

An application of this function for computing x^n needs n multiplications. Despite this lack of efficiency, and thanks to its simplicity, we keep it as a specification for more efficient and complex exponentiation algorithms. A function will be considered a *correct* exponentiation function if we can prove it is extensionally equivalent to `power`.

10.2.1 A logarithmic exponentiation function

Using the following equations, we can easily define a polymorphic exponentiation whose application requires only a logarithmic number of multiplications.

$$x^1 = x \tag{10.1}$$
$$x^{2p} = (x^2)^p \tag{10.2}$$
$$x^{2p+1} = (x^2)^p \times x \tag{10.3}$$
$$x^1 \times a = x \times a \tag{10.4}$$
$$x^{2p} \times a = (x^2)^p \times a \tag{10.5}$$
$$x^{2p+1} \times a = (x^2)^p \times (a \times x) \tag{10.6}$$

In equalities 10.4 to 10.6, the variable a plays the role of an *accumulator* whose initial value (set by 10.3) is x. This accumulator helps us to get a tail-recursive implementation.

For instance, the computation of 2^{14} can be decomposed as follows:

$$\begin{aligned} 2^{14} &= 4^7 \\ &= 16^3 \times 4 \\ &= 256^1 \times (4 \times 16) \\ &= 16384 \end{aligned}$$

With the same notations as in Sect 10.2 on the previous page, we can implement this algorithm in Gallina. The following definitions are still within the scope of the section open in 10.2 on the preceding page.

From Module additions.FirstSteps

```
Fixpoint binary_power_mult (x a:A)(p:positive) : A
  :=
  match p with
    | xH =>  a * x
    | xO q => binary_power_mult  (x * x) a q
    | xI q =>  binary_power_mult  (x * x) (a * x) q
  end.

Fixpoint Pos_bpow (x:A)(p:positive) :=
 match p with
  | xH => x
  | xO q => Pos_bpow  (x * x) q
  | xI q => binary_power_mult   (x * x) x q
end.

Definition N_bpow x (n:N) :=
  match n with
  | 0%N => 1
  | Npos p => Pos_bpow x p
  end.

End Definitions.
```

Let us close the section `Definitions` and mark the argument `A` as implicit.

```
End Definitions.

Arguments N_bpow {A}.
Arguments power {A}.
```

Remark 10.1 Our function `Pos_bpow` can be considered as a tail recursive variant of the following function defined in `Coq.PArith.BinPosDef`.

```
Definition iter_op {A}(op:A->A->A) :=
  fix iter (p:positive)(a:A) : A :=
  match p with
    | 1 => a
    | p~0 => iter p (op a a)
    | p~1 => op a (iter p (op a a))
  end.
```

This scheme is used in `Coq.ZArith.Zpow_alt` in order to define a logarithmic exponentiation `Zpower_alt` on `Z` (notation : x `^^` p).

Remark Note that closing the section `Definitions` makes us lose the handy notations `_ * _` and `one`. Fortunately, *operational type classes* will help us to define nice infix notations for polymorphic functions (Sect. 10.3.1 on page 200).

10.2.2 Examples of computation

It is now possible to test our functions with various interpretations of $\times$ and 1:

```
Compute N_bpow Z.mul 1%Z 2%Z 10.
```

```
 = 1024%Z
     : Z
```

```
Require Import String.
Open Scope string_scope.

Compute N_bpow append "" "ab"  12.
```

```
  = "abababababababababababab"
     : string
```

10.2.2.1 Exponentiation on 2×2 matrices

Our second example is a definition of M^n where M is a 2×2 matrix over any "scalar" type A, assuming one can provide A with a semi-ring structure [The].

A 2×2 matrix will be simply represented by a structure with four fields; each field `c`ij is associated with the i-th line and j-th column of the considered matrix.

```
Module M2.
Section Definitions.

  Variables (A: Type) (zero one : A)  (plus mult  : A -> A -> A).

  Variable rt : semi_ring_theory  zero one plus mult   (@eq A).
  Add Ring Aring : rt.

  Notation "0" := zero.
  Notation "1" := one.
  Notation "x + y" := (plus x y).
  Notation "x * y " := (mult x y).

  Structure t : Type := mat{c00 : A;  c01 : A;  c10 : A;  c11 : A}.
```

The structure type `M2.t` allows us to define the product of two matrices.

```
Definition M2_mult (M M':t) : t := mat
  (c00 M * c00 M' + c01 M * c10 M') (c00 M * c01 M' + c01 M * c11 M')
  (c10 M * c00 M' + c11 M * c10 M') (c10 M * c01 M' + c11 M * c11 M').
```

The neutral element for `M2_mult` is the identity matrix.

```
Definition Id2 : t := mat 1 0 0 1.

End M2_Definitions.
End M2.
```

10.2.3 Computing Fibonacci numbers

The sequence of Fibonacci numbers is defined by the following equations:

$$F_0 = 1 \tag{10.7}$$
$$F_1 = 1 \tag{10.8}$$
$$F_n = F_{n-1} + F_{n-2} \quad (n \geq 2) \tag{10.9}$$

In Coq, one can define this function by simple recursion.
From Library additions.Fib2

```
Fixpoint fib (n:nat) : N :=
  match n with
    0%nat | 1%nat => 1%N
  | S (S p as q) => fib p + fib q
  end.

Compute fib 20.
```

```
= 10946 : N
```

In [BC04], several exercises [1] present ways to compute Fibonacci numbers, with the less number of recursive calls as possible. Please note that these optimizations and the formal proof of their correctness are *ad-hoc*, *i.e.*, exclusively written for the Fibonacci numbers. In contrast, the optimizations we present in this document apply, in their vast majority, *generic* techniques of efficient computation of powers in a monoid. This example of Fibonacci numbers has been developed with Yves Bertot, who wrote a first version with `SSreflect/Mathcomp` [MT18].

10.2.3.1 Using 2x2 integer matrices

The following properties are well known. They are left as an exercise, since they are not part of our development.

Exercise 10.1 1. Prove in Coq the following equality (for any $n \geq 2$).

$$\begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix} \begin{pmatrix} F_n & F_{n-1} \\ F_{n-1} & F_{n-2} \end{pmatrix} = \begin{pmatrix} F_{n+1} & F_n \\ F_n & F_{n-1} \end{pmatrix}$$

2. Infer (still in Coq) the following equality (still for $n \geq 2$).

$$\begin{pmatrix} F_n & F_{n-1} \\ F_{n-1} & F_{n-2} \end{pmatrix} = \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}^n$$

3. Write a function using the previous equality for computing the n-th Fibonacci number, and prove its equivalence with `fib`.

[1] Exercises 9.8 (page 270), 9.10 (page 271), 9.15 (page 276), 9.17 (page 284), and 15.8 (page 418).

10.2.3.2 Removing duplicate computations

Yves Bertot's optimization relies on the observation that all the powers of $\begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}$ have the form $\begin{pmatrix} a+b & a \\ a & b \end{pmatrix}$ where a and b are natural numbers.

Thus, it is possible to remove duplicate data and computations by reflecting matrix multiplication and identity into $\mathbb{N} \times \mathbb{N}$.

If we pose $\varphi(a,b) = \begin{pmatrix} a+b & a \\ a & b \end{pmatrix}$, then $\varphi(a,b) \times \varphi(c,d) = \varphi(ac + ad + bc, ac + bd)$, and $\varphi(0,1) = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}$.

Exercise 10.2 Prove formally these properties. *Please note that their proof is not needed in our development, they just help to understand the following optimization.*

So, let us define a binary operation, which makes $\mathbb{N} \times \mathbb{N}$ a monoid (with $(0,1)$ as neutral element).

From Library additions.Fib2

The `Monoid` type class is defined page 202.

```
Definition mul2 (p q : N * N) :=
  match p, q with (a, b),(c,d) => (a*c + a*d + b*c, a*c + b*d) end.

Instance Mul2 : Monoid  mul2 (0,1).
(* Proof omitted *)
```

The following lemma is a simplification of the equality 1 on the preceding page.

```
Lemma next_fib (n:nat) : mul2 (1,0) (fib (S n), fib n) =
                         (fib (S (S n)), fib (S n)).
```

By induction[2] over n, we obtain a the following equalities.

```
Definition fib_mul2 n := let (a,b) := power (M:=Mul2) (1,0) n   in (a+b).

Lemma fib_mul2_OK_0 (n:nat) : power (M:=Mul2) (1,0) (S (S n)) =
                              (fib (S n), fib n).

Lemma fib_mul2_OK n : fib n = fib_mul2 n.
```

Thus, any function able to compute more or less efficiently powers in a monoid will give an algorithm for computing Fibonacci numbers. Unlike the *ad-hoc* aforementioned proofs of [BC04], the correctness of such an algorithm is a direct consequence of the correctness of the used powering function. Several examples will be presented in the rest of this document (in sections 10.4.6 on page 212).

To do 10.1 *Document the files contributed by Yves*

[2] Simple induction, since n has type `nat`.

- *additions/fib.v (to rename ?)*
- *additions/stub.ml (to keep inside theories/ or move to src/ ?)*
- *theories/additions/make_fib_tests.txt (to put in a Makefile?)*

10.2.4 Formal specification of an exponentiation function: a first attempt

Let us compare the functions `power` and `N_bpow`. The first one is obviously correct, since it is a straightforward translation of the mathematical definition. The second one is much more efficient, but it is not obvious that its 18-line long definition is bug-free. Thus, we must prove that the two functions are extensionally equal (taking into account conversions between `N` and `nat`).

More abstractly, we can define a predicate that characterizes any correct implementation of `power`, this "naive" function being a *specification* of any polymorphic exponentiation function.

First, we define a type for any such function.

```
Definition power_t := forall (A:Type)
                             (mult : A -> A -> A)
                             (one:A)
                             (x:A)
                             (n:N), A.
```

Then, we would say that a function `f:power_t` is a correct exponentiation function if it is extensionally equal to `power`.

```
Module Bad.

 Definition correct_expt_function (f : power_t) : Prop :=
  forall A (mult : A -> A -> A) (one:A)
            (x:A) (n:N),
            power mult one x (N.to_nat n) = f A mult one x n.
```

Unfortunately, our definition of `correct_expt` is too general. It suffices to build an interpretation where the multiplication is not associative or `one` is not a neutral element to obtain different results through the two functions.

```
Section CounterExample.
    Let mul (n p : nat) := n + 2 * p.
    Let one := 0.

    Remark mul_not_associative :
      exists  n p q,  mul n (mul p q) <> mul (mul n p) q.
    Proof.
        exists 1, 1, 1; discriminate.
    Qed.

    Remark one_not_neutral  :
      exists n : nat, mul one n <> n.
    Proof.
```

```
      exists 1; discriminate.
    Qed.

    Lemma correct_expt_too_strong :
          ~ correct_expt_function (@N_bpow).
    Proof.
      intro H; specialize (H _ mul one 1  7%N).
      discriminate H.
    Qed.

End CounterExample.
End Bad.
```

So, we will have to improve our definition of correctness, by restricting the universal quantification to associative operations and neutral elements, *i.e.*, by considering *monoids*. An exponentiation function will be considered as correct if it returns always the same result as `power` *in any monoid*.

10.3 Representing monoids in Coq

In this section, we present a "minimal" algebraic framework in which exponentiation can be defined and efficiently implemented.

Exponentiation is built on multiplication, and many properties of this operation are derived from the associativity of multiplication. Furthermore, if we allow the exponent to be any natural number, including 0, then we need to consider a neutral element for multiplication.

The structure on which we define exponentiation is called a *monoid*. It is composed of a *carrier* A, an associative binary operation $\times$ on A, and a neutral element $\mathbb{1}$ for $\times$. The required properties of $\times$ and $\mathbb{1}$ are expressed by the following equations:

$$\forall x\, y\, z : A,\ x \times (y \times z) = (x \times y) \times z \tag{10.10}$$

$$\forall x : A,\ x \times \mathbb{1} = \mathbb{1} \times x = x \tag{10.11}$$

In Coq, we define the monoid structure in terms of *type classes*[SO08, SvdW11]. The tutorial on type classes [CS] gives more details on type classes and operational type classes, also illustrated with the monoid structure.

First, we define a class and a notation for representing multiplication operators, then we use these definitions for defining the `Monoid` type class.

10.3.1 A common notation for multiplication

Operational type classes [SvdW11] allow us to define a common notation for multiplication in any algebraic structure. First, we associate a class to the notion of *multiplication* on any type A.

From Module additions/Monoid_def.v.

```
Class Mult_op (A:Type) := mult_op : A -> A -> A.
```

From the type theoretic point of view, the term (`Mult_op A`) is $\beta\delta$-reducible to $A \to A \to A$, and if *op* has type (`Mult_op A`), then (`@mult_op A` *op*) is convertible with *op*.

We are now ready to define a new notation scope, in which the notation `x * y` will be interpreted as an application of the function `mult_op`.

```
Delimit Scope M_scope with M.
Infix "*" := mult_op : M_scope.
Open Scope M_scope.
```

Let us show two examples of use of the notation scope `M_scope`. Each example consists in declaring an instance of `Mult_op`, then type checking or evaluating a term of the form `x * y` in `M_scope`.

Note that, since the reserved notation `"_ * _ "` is present in several scopes such as `nat_scope`, `Z_scope`, `N_scope`, etc., in addition to `M_scope`, the user should take care of which scopes are active — and with which precedence — in a Gallina term. In case of doubt, explicit scope delimiters should be used.

10.3.1.1 Multiplication on Peano numbers

Multiplication on type `nat`, called `Nat.mul` in Standard Library, has type `nat -> nat -> nat`, which is convertible with `Mult_op nat`. Thus the following definition is accepted:

```
Instance nat_mult_op : Mult_op nat  := Nat.mul.
```

Inside `M_scope`, the expression `3 * 4` is correctly read as an application of `mult_op`. Nevertheless this term is convertible with `Nat.mul 3 4`, as shown by the interaction below.

From Module additions.Monoid_def

```
Set Printing All.
Check  3 * 4.
```

```
@mult_op nat nat_mult_op (S (S (S O))) (S (S (S (S O))))
     : nat
```

```
Unset Printing All.
Compute 3 * 4.
```

```
 = 12  : nat
```

10.3.1.2 String concatenation

We can use the notation `"_ * _ "` for other types than numbers. In the following example, the expression `"abc" * "def"` is interpreted as `@mult_op string ?X "abc" "def"`, then the type class mechanism replaces the unknown ?X with `string_op`.

From Module additions.Monoid_def

```
Require Import String.

Instance string_op : Mult_op string := append.
Open Scope string_scope.

Example ex_string : "ab" * "cde" = "abcde".
Proof. reflexivity. Qed.
```

10.3.1.3 Solving ambiguities

Let A be some type, and let us assume there are several instances of `Mult_op` A. For solving ambiguity issues, one can add a *precedence* to each instance declaration of `Mult_op` A. In any case, such ambiguity can be addressed by explicitly providing some arguments of `mult_op`. For instance, in Sect. 10.3.3.2 on the facing page, we consider various monoids on types `nat` and `N`.

10.3.2 The Monoid type class

We are now ready to give a definition of the `Monoid` class, using `*` as an infix operator in scope %M for the monoid multiplication.

The following class definition, from Module additions.Monoid_def, is parameterized with some type A, a multiplication (called `op` in the definition), and a neutral element $\mathbb{1}$ (called `one` in the definition).

```
Class Monoid {A:Type}(op : Mult_op A)(one : A) : Prop :=
{
    op_assoc : forall x y z:A, x * (y * z) = x * y * z;
    one_left : forall x, one * x = x;
    one_right : forall x, x * one = x
}.
```

10.3.3 Building instances of `Monoid`

Let A be some type, *op* an instance of `Mult_op` A and *one:* A. In order to build an instance of (`Monoid` A *op one*), one has to provide proofs of "monoid axioms" `op_assoc`, `one_left` and `one_right`.

Let us show various instances, which will be used in further proofs and examples. Complete definitions and proofs are given in File additions/Monoid_instances.v.

10.3.3.1 Monoid on `Z`

The following monoid allows us to compute powers of integers of arbitrary size, using type `Z` from standard library:

```
Instance Z_mult_op : Mult_op Z := Z.mul.

Instance ZMult : Monoid  Z_mult_op 1.
Proof.
  split.
```

```
3 subgoals, subgoal 1 (ID 8)

  ============================
   forall x y z : Z, (x * (y * z))%M = (x * y * z)%M

subgoal 2 (ID 9) is:
 forall x : Z, (1 * x)%M = x
subgoal 3 (ID 10) is:
 forall x : Z, (x * 1)%M = x}
```

```
    all: unfold Z_mult_op, mult_op;intros;ring.
Qed.
```

10.3.3.2 Monoids on type nat and N

We define two monoids on type `nat`:

- The "natural" monoid $(\mathbb{N}, \times, 1)$:

```
Instance nat_mult_op : Mult_op nat | 5 := Nat.mul.

Instance  Natmult : Monoid nat_mult_op  1%nat | 5
Proof.
   split;unfold nat_mult_op, mult_op; intros; ring.
Qed.
```

- The "additive" monoid $(\mathbb{N}, +, 0)$. This monoid will play an important role in correctness proofs of complex exponentiation algorithms. Its most important property is that the n-th power of 1 is equal to n. See Sect. 10.7.4 on page 225 for more details.

```
Instance nat_plus_op : Mult_op nat | 12 := Nat.add.

Instance Natplus : Monoid nat_plus_op  0%nat | 12.
(* Proof omitted *)
```

Similarly, instances `NMult` and `NPlus` are built for type `N`, and `PMult` for type `positive`.

10.3.3.3 Machine integers

Cyclic numeric types are good candidates for testing exponentiations with big exponents, since the size of data is bounded.

The type `int31` is defined in Module **Coq.Numbers.Cyclic.Int31.Int31** of Coq's standard library. The tactic `ring` works with this type, and helps us to register an instance `Int31Mult` of class `Monoid int31_mult_op 1`.

```
Instance int31_mult_op : Mult_op int31 := mul31.

Instance  Int31mult : Monoid int31_mult_op  1.
```

```
Proof.
   split;unfold int31_mult_op, mult_op; intros; ring.
Qed.
```

Beware that machine integers are not natural numbers!

```
Module Bad.

Fixpoint int31_from_nat (n:nat) :=
  match n with
  | O => 1
  | S p => 1 + int31_from_nat p
  end.

Coercion int31_from_nat : nat >-> int31.

Fixpoint fact (n:nat) :=
  match n with
   | O => 1
   | S p => n * fact p
  end.

Example fact_zero : exists n:nat, fact n = 0.
Proof.  now exists 40%nat.  Qed.

End Bad.
```

10.3.4 Matrices on a semi-ring

In Sect.10.2.2.1 on page 196, we defined a function for computing powers of any 2×2 matrix over any semi-ring. For proving a simple property of matrix exponentiation, we had to prove that matrix multiplication is associative and admits the identity matrix as a neutral element. These properties are easily expressed within the type class framework, by defining a *family* of monoids. It suffices to define an instance of `Monoid` within the scope of a hypothesis of type `semi_ring_theory`.

```
Section M2_def.
Variables (A:Type)
          (zero one : A)
          (plus mult  : A -> A -> A).

 Variable rt : semi_ring_theory  zero one plus mult  (@eq A).
 Add  Ring Aring : rt.
```

```
Structure M2 : Type := {c00 : A;  c01 : A;
                        c10 : A;  c11 : A}.

Definition Id2 : M2 := Build_M2 1 0 0 1.

Definition M2_mult (m m':M2) : M2 :=
 Build_M2
```

```
          (c00 m * c00 m' + c01 m * c10 m')
          (c00 m * c01 m' + c01 m * c11 m')
          (c10 m * c00 m' + c11 m * c10 m')
          (c10 m * c01 m' + c11 m * c11 m').

Global Instance M2_op : Mult_op M2 := M2_mult.
```

```
Global Instance M2_Monoid : Monoid   M2_op Id2.
(* Proof omitted *)

End M2_def.

Arguments M2_Monoid {A zero one plus mult} rt.
```

10.3.5 Monoids and equivalence relations

In some contexts, the "axioms" of the `Monoid` class may be too restrictive. For instance, consider multiplication in $\mathbb{Z}/m\mathbb{Z}$ where $1 < m$. Although it could be possible to compute with values of the dependent type `{n:N | n < m}`, it looks simpler to compute with numbers of type `N` and consider the multiplication $x \times y \mod m$.

It is easy to prove that this operation is associative, using library `NArith`. Unfortunately, the following proposition is false in general (left as an exercise).

$$\forall x : N, (1 * x) \mod m = x$$

Thus, we define a more general class, parameterized by an equivalence relation `Aeq` on a type `A`, compatible with the multiplication `*`. The laws of associativity and neutral element are not expressed as Leibniz equalities but as equivalence statements:

First, let us define an operational type class for equivalence relations:

From Module additions.Monoid_def

```
Class Equiv A := equiv : relation A.

Infix "==" := equiv (at level 70) : type_scope.
```

The definition of class `EMonoid` looks like `Monoid`'s definition, plus some constraints on `E_eq`.

Please look for instance at our tutorial on type classes and relations [CS] for understanding the use of type classes `Equivalence`, `Reflexive`, `Proper`, etc, in relation with tactics like `rewrite`, `reflexivity`, etc., in proofs which involve equivalence relations instead of equality.

```
Class EMonoid (A:Type)(E_op : Mult_op A)(E_one : A)
      (E_eq: Equiv A): Prop :=
  {
    Eq_equiv :> Equivalence equiv;
    Eop_proper :> Proper (equiv ==> equiv ==> equiv) E_op;
    Eop_assoc : forall x y z, x * (y * z) == x * y * z;
    Eone_left : forall x,  E_one * x == x;
```

```
    Eone_right : forall x,  x * E_one ==  x
  }.
```

10.3.5.1 Coercion from Monoid to EMonoid

Every instance of class `Monoid` can be transformed into an instance of `EMonoid`, considering Leibniz' equality `eq`. Thus, our definitions and theorems about exponentiation will take place as much as possible within the more generic framework of `EMonoid`s.

```
Global Instance eq_equiv {A} : Equiv A := eq.

Global Instance Monoid_EMonoid `(M:@Monoid A op one) :
        EMonoid  op one eq_equiv.
Proof.
split; unfold eq_equiv, equiv in *.
 - apply eq_equivalence.
 - intros x y H z t H0; now subst.
 - intros; now rewrite (op_assoc).
 - intro; now rewrite one_left.
 - intro; now rewrite one_right.
Defined.
```

Remark 10.2 In the definition of `Monoid_EMonoid`, the free variables `A`, `op` and `one` are automatically generalized thanks to the *backquote* syntax (see the section about implicit generalization in the reference manual [The]).

Thanks to the following *coercion*, every instance of `Monoid` can now be considered as an instance of `EMonoid`. For more details, please look at the section *Implicit Coercions* of Coq's reference manual [The].

```
Coercion Monoid_EMonoid : Monoid >-> EMonoid.
```

From Module additions.Monoid_instances

```
Check NMult : EMonoid  N.mul 1%N eq.
```

```
NMult:EMonoid N.mul 1%N eq
   : EMonoid N.mul 1%N eq
```

10.3.5.2 Example : Arithmetic modulo m

The following instance of `EMonoid` describes the set of integers modulo m, where m is any integer greater than or equal to 2. For simplicity's sake, we represent such values using the `N` type, and consider "equivalence modulo m" instead of equality. Note that the law of associativity has been stated as Leibniz' equality.

```
Section Nmodulo.
  Variable m : N.
  Hypothesis m_gt_1 : 1 < m.
```

```
  Definition mult_mod ( x y : N) := (x * y) mod m.
  Definition mod_eq ( x y: N) := x mod m = y mod m.

  Global Instance mod_equiv : Equiv N := mod_eq.

  Global Instance mod_op : Mult_op N := mult_mod.

  Global Instance mod_Equiv : Equivalence mod_equiv.
  (* Proof omitted *)

  Global Instance mult_mod_proper :
  Proper (mod_equiv ==> mod_equiv ==> mod_equiv)  mod_op.
  (* Proof omitted *)

  Local Open Scope M_Scope.

  Lemma mult_mod_associative :
  forall x y z,  x * (y * z) = x * y * z.
  (* Proof omitted *)

  Lemma one_mod_neutral_l  : forall x, 1 * x ==  x.
  (* Proof omitted *)

  Lemma one_mod_neutral_r  : forall x, x * 1 == x.
  (* Proof omitted *)

  Global Instance Nmod_Monoid : EMonoid  mod_op 1 mod_equiv.
  (* Proof omitted *)

End Nmodulo.
```

10.3.5.2.1 Example In the following interaction, we show how to instantiate the parameter m to a concrete value, for instance 256.

```
Section S256.
Let mod256 :=  mod_op 256.
Local Existing Instance mod256 | 1.

Compute (211 * 67)
```

```
= 57 : N
```

```
End S256.
```

Outside the section `S256`, the term `(211 * 67)%M` is interpreted as a plain multiplication in type `N`:

```
Compute (211 * 67)%M.
```

```
= 14137 : N
```

10.4 Computing powers in any EMonoid

The module additions.Pow defines two functions for exponentiation on any `EMonoid` on carrier A. They are essentially the same as in Sect. 10.2 on page 193. The main difference lies in the arguments of the functions, which now contain an instance `M` of class `EMonoid`. Thus, the arguments associated with the multiplication, the neutral element and the equivalence relation associated with `M` are left implicit.

10.4.1 The naive (linear) algorithm

The new version of the linear exponentiation function is as follows:

```
Fixpoint power`{M: @EMonoid A  E_op E_one E_eq}
               (x:A) (n:nat) :=
match n with
| 0%nat => E_one
| S p =>   x * x ^ p
end
where "x ^ n" := (power x n) : M_scope.
```

The three following lemmas will be used by the `rewrite` tactic in further correctness proofs. Note that the first two lemmas are strong (*i.e.*, Leibniz) equalities, whilst `power_eq3` is only an equivalence statement, because its proof uses one of the `EMonoid` laws, namely `Eone_right`.

```
Lemma power_eq1 {A:Type} `{M: @EMonoid A  E_op E_one E_eq}
               (x:A) :  x ^ 0 = E_one.
Proof. reflexivity. Qed.

Lemma power_eq2 {A:Type} `{M: @EMonoid A  E_op E_one E_eq}
                (x:A) (n:nat) :
                x ^ (S n)  = x * x ^ n.
Proof. reflexivity. Qed.

Lemma power_eq3 {A:Type} `{M: @EMonoid A  E_op E_one E_eq}
                (x:A) : x ^ 1 == x.
Proof. cbn; rewrite Eone_right; reflexivity. Qed.
```

10.4.1.1 Examples of computation

In the following computations, we first show an exponentiation in $\mathbb{Z}$, then in the type of 31-bit machine integers.[3]

From Module additions.Demo_power

```
Open Scope M_scope.

Compute 22%Z ^ 20.
```

[3] `phi` and `phi_inv` are standard library's conversion functions between types `Z` and `int31`, used for making it possible to read and print values of type `int31`.

```
= 705429498686404044207947776%Z
```

```
Import Int31.
Coercion phi_inv : Z >-> int31.

Compute (22%int31 ^ 20).
```

```
   = 2131755008%int31
     : int31
```

10.4.2 The binary exponentiation algorithm

Please find below the implementation of binary exponentiation using type classes (to be compared with the version in 10.2.1 on page 194).

From Module additions.Pow

```
Fixpoint binary_power_mult `{M: @EMonoid A E_op E_one E_eq}
             (x a:A)(p:positive) : A
  :=
  match p with
    | xH =>    a * x
    | xO q => binary_power_mult  (x * x) a q
    | xI q =>   binary_power_mult (x * x) (a * x) q
  end.

Fixpoint Pos_bpow  `{M: @EMonoid A E_op E_one E_eq}
         (x:A)(p:positive) :=
 match p with
  | xH => x
  | xO q => Pos_bpow  (x * x) q
  | xI q => binary_power_mult (x * x) x q
end.
```

It is easy to extend `Pos_bpow`'s domain to the type of all natural numbers:

From Module additions.Pow

```
Definition N_bpow {A} `{M: @EMonoid A E_op E_one E_eq} x (n:N) :=
  match n with
  | 0%N => E_one
  | Npos p => Pos_bpow x p
  end.

Infix "^b" := N_bpow (at level 30, right associativity): M_scope.
```

10.4.3 Refinement and correctness

We have got two functions for computing powers in any monoid. So, it is interesting to ask oneself whether this duplication is useful, and which would be the respective role of `N_bpow` and `power`.

- The function `power`, although very inefficient, is a direct translation of the mathematical definition, as shown by lemmas `power_eq1` to `power_eq3`. Moreover, its structural recursion over type `nat` allows simple proofs by induction over the exponent. Thus, we will consider `power` as a *specification* of any exponentiation algorithm.

- Functions `N_bpow` and `Pos_bpow` are more efficient, but less readable than `power`, and we cannot use these functions before having proved their correctness. In fact, the correctness of `N_bpow` and `Pos_bpow` will mean "being extensionally equivalent to `power`". For instance `N_bpow`'s correctness is expressed by the following statement (in the context of an `EMonoid` on type `A`).

  From Module additions.Pow

```
Lemma N_bpow_ok :
forall (x:A) (n:N),   x ^b n  == x ^ N.to_nat n.
(* Proof omitted *)
```

The relationship between `power` and `N_bpow` can be considered as a kind of *refinement* as in the B-method [Abr96]. Note that the two representations of natural numbers and the function `N.to_nat` form a kind of *data refinement* [Abr10, CDM13].

10.4.4 Proof of correctness of binary exponentiation w.r.t. the function `power`

Section `M_given` of Module additions.Pow is devoted to the proof of properties of the functions above. Note that properties of `power` refer to the *specification* of exponentiation, and can be applied for proving correctness of any implementation.

In this section, we consider an arbitrary instance `M` of class `EMonoid`.

```
Section M_given.
 Variables (A:Type) (E_op : Mult_op A)(E_one:A) (E_eq : Equiv A).
 Context (M:EMonoid  E_op E_one E_eq).
```

10.4.4.1 Properties of exponentiation

We establish a few well-known properties of exponentiation, and define some basic tactics for simplifying proof search.

```
Ltac monoid_rw :=
    rewrite Eone_left  ||
    rewrite Eone_right  ||
    rewrite Eop_assoc .

Ltac monoid_simpl := repeat monoid_rw.

Section About_power.
```

In order to make possible proof by rewriting on expressions which contain the exponentiation operator, we have to prove that, whenever x `==` y, the equality x^n `==` y^n holds for any exponent n. For this purpose, we use the `Proper` class of module Coq.Classes.Morphisms

```
Global Instance power_proper :
     Proper (equiv ==> eq ==> equiv) power.
(* Proof omitted *)
```

In the following proofs, we note how notations, type classes and generalized rewriting can be used to write algebraic properties in a nice way.

```
Lemma power_of_plus :   forall x n p, x ^ (n + p) ==  x ^ n *  x ^ p.
(* Proof omitted *)

Ltac power_simpl :=
    repeat (monoid_rw || rewrite <- power_x_plus).
```

Please note that the following two lemmas *do not require* the operation `*` to be commutative.

```
Lemma power_commute :
    forall x n p, x ^ n * x ^ p ==  x ^ p * x ^ n.
(* Proof omitted *)

Lemma power_commute_with_x :
    forall x n,  x * x ^ n == x ^ n * x.
(* Proof omitted *)

Lemma power_of_power :
   forall x n p,  (x ^ n) ^ p == x ^ (p * n).
(* Proof omitted *)
```

The following two equalities are auxiliary lemmas for proving correctness of the binary exponentiation functions.

```
Lemma sqr_def : forall x, x ^ 2 ==  x * x.
(* Proof omitted *)

Lemma power_of_square :
  forall x n, (x * x) ^ n ==  x ^ n * x ^ n.
(* Proof omitted *)
```

10.4.5 Equivalence of the two exponentiation functions

Since `binary_power_mult` is defined by structural recursion on the exponent `p:positive`, its basic properties are proved by induction along `positive`'s constructors.

From Module additions.Pow

```
Lemma binary_power_mult_ok :
  forall p a x,   binary_power_mult  x a p  ==
                  a * x ^ Pos.to_nat p.
Proof.
  induction p as [q IHq | q IHq| ].
 (* Rest of proof omitted *)
```

```
Lemma Pos_bpow_ok :
  forall p x, Pos_bpow x p == x ^ Pos.to_nat p.
(* Proof omitted *)

Lemma N_bpow_ok :
  forall n x,  x ^b n  == x ^ N.to_nat n.
(* Proof omitted *)
```

```
Lemma N_bpow_ok_R :
  forall n x, x ^b (N.of_nat n)   ==  x ^  n.
(* Proof omitted *)

Lemma Pos_bpow_ok_R :
   forall p x, p <> 0 ->
                      Pos_bpow x  (Pos.of_nat p)   ==  x ^  p.
(* Proof omitted *)

End About_power.
```

10.4.5.1 Remark

The preceding lemmas can be applied for deriving properties of the binary exponentiation functions:

```
Lemma N_bpow_commute : forall x n p,
                         x ^b n *  x ^b p ==
                         x ^b p *  x ^b n.
Proof.
 intros x n p; repeat rewrite N_bpow_ok.
 rewrite power_commute; reflexivity.
Qed.
```

10.4.6 Fibonacci, once again

We can use the function `Pos_bpow` for computing Fibonacci numbers (see Section 10.2.3.2 on page 198).

```
Definition fib_pos n :=
  let (a,b) := Pos_bpow (M:= Mul2) (1,0) n in
  (a+b).

Time Compute fib_pos 153%positive.
```

```
  68330027629092351019822533679447
     : N
Finished transaction in 0.002 secs (0.002u,0.s) (successful)
```

Fibonacci will come back in Sect. 10.9.10 on page 255.

10.5 Comparing exponentiation algorithms with respect to efficiency

It looks obvious that the binary exponentiation algorithm is more efficient than the naive one. Can we study *within Coq* the respective efficiency of both functions? Let us take a simple example with the exponent 17, in any `EMonoid`.

```
Eval simpl in   fun (x:A) => x ^b 17.
```

```
 = fun x : A =>
       x *
       (x * x * (x * x) * (x * x * (x * x)) *
        (x * x * (x * x) * (x * x * (x * x))))
     : A -> A
```

Therefore, we note that the term (`fun (x:A) =>x ^b 17`) is convertible, — *thus logically indistinguishable* —, with a function that performs 16 multiplications.

Likewise, let us simplify the term (`fun (x:A) =>x ^ 17`):

```
Eval simpl in   fun x =>  x ^ 17.
```

```
= fun x : A =>
   x * (x *  (x *  (x *  (x *   (x *  (x *  (x *
    (x * (x * (x * (x * (x * (x * (x * (x * (x * one)))))
  ))))))))))
```

From these tests, we may infer that representing exponentiation algorithms as Coq functions hides information about the real structure of the computations, particularly the sharing on intermediate computations.

Thus, we propose to define a data structure that makes explicit the sequence of multiplications that lead to the computation of x^n. For instance, the values of `x * x` and `x * x * (x * x)` are used twice in the computation of x^{17} with the binary algorithm. This information should appear explicitly in the data structure chosen for representing exponentiation algorithms.

It is well known that local variables can be used to store intermediate results. In an `ISWIM - ML` style, the function computing x^{17} could be written as follows:

```
Definition pow_17  (x:A) :=
  let x2 := x * x in
  let x4 := x2 * x2 in
```

```
  let x8 := x4 * x4 in
  let x16 := x8 * x8 in
  x16 * x.
```

Unfortunately, Coq's **let-in** construct is useless for our purpose, since ζ-conversion would make the sharing of computations disappear.

```
Eval cbv  zeta beta delta [pow_17]  in  pow_17.
```

```
= fun x : A =>
      x * x * (x * x) * (x * x * (x * x)) *
      (x * x * (x * x) * (x * x * (x * x))) * x
    : A -> A
```

In the next section, we propose to use a *data structure* for representing the computations that lead to the evaluation of some power x^n, where intermediary results are explicitly named for further use in the rest of the computation.

10.6 Addition chains

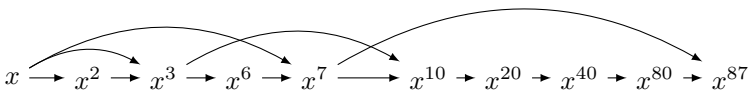
An *addition chain* (in short, a *chain*) [Bra39] is a representation of a sequence of intermediate steps that lead to the evaluation of x^n, under the assumption that each of these steps is a computation of a power x^i, with $i < n$.

In articles from the combinatorist community, *e.g.*, [Bra39, BB87], addition chains are represented as sequences of positive integers, each member of which is either 1 or the sum of two previous elements. For instance, the three following sequences are addition chains for the exponent 87:

$$c_{87} = (1, 2, 3, 6, 7, 10, 20, 40, 80, 87) \tag{10.12}$$
$$c'_{87} = (1, 2, 3, 4, 7, 8, 16, 23, 32, 64, 87) \tag{10.13}$$
$$c''_{87} = (1, 2, 4, 8, 16, 32, 64, 80, 84, 86, 87) \tag{10.14}$$

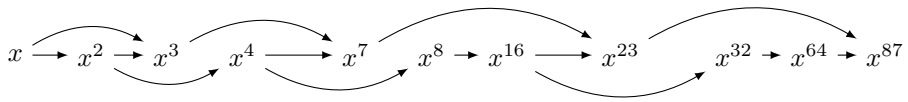
It is possible to associate to any addition chain a directed acyclic graph: whenever $i = j + k$, there is an arc from x^j to x^i and an arc from x^k to x^i. Figures 10.1 and 10.2 show the graphical representations of c_{87} and c'_{87}. Please note that some chains may be represented by various different dags (directed acyclic graphs). For instance, we can associate four different dags to the chain $(1, 2, 3, 4, 6, 9, 13)$.

Figure 10.1: Graphical representation of c_{87} (9 multiplications)



Let us assume that the efficiency of an exponentiation algorithm is proportional to the number of multiplications it requires. This assumption looks

Figure 10.2: Graphical representation of c'_{87} (10 multiplications)



reasonable when the data size is bounded (for instance : machine integers, arithmetic modulo m, etc.). Let us define the *length* of a chain c as its number $|c|$ of exponents (without counting the initial 1). This length is the number of multiplications needed for computing the x^is by applying the following algorithm:

> For any item i of c, there exists j and k in c, where $i = j + k$, and x^j and x^k are already computed.
>
> Thus, compute $x^i = x^j \times x^k$.

In our little example, we have $|c_{87}| = 9 < 10 = |c'_{87}|$. In the rest of this chapter, we will try to focus on the following aspects:

- Define a representation of addition chains that allows to compute efficiently x^n in any monoid, for quite large exponents n;
- Certify that our representation of chains is correct, *i.e.*, determines a computation of x^n for a given n;
- Define and certify functions for automatically generating correct and shortest as possible chains.

In a previous work [BCHM95, BCS91, Cas04], addition chains were represented so as to allow efficient computations of powers and certification of a family of automatic chain generators. We present here a new implementation that takes into account some advances in the way we use Coq: generalized rewriting, type classes, parametricity, etc.

10.6.1 A type for addition chains

Let us recall that we want to represent some algorithms of the form described in section 10.5, but avoiding to represent intermediate results by **let-in** constructs. We describe below the main design choices we made:

- Continuation Passing Style (CPS) [Rey93] is a way to make explicit the control in the evaluation of an expression, in a purely functional way. For every intermediate computation step, the result is sent to a *continuation* that executes the further continuations. When the continuation is a lambda-abstraction, its bound variable gives a *name* to this result
- Like in Parametric Higher Order Abstract Syntax (PHOAS) [Chl08], the local variables associated to intermediate results are represented by variables of type A, where A is the underlying type of the considered monoid.

10.6.1.1 Definition

Let `A` be some type; a *computation* on `A` is

- either a final step, returning some value of type `A`
- or the multiplication of two values of type `A`, with a *continuation* that takes as argument the result of this multiplication, then starts a new computation.

In the following inductive type definition, the intended meaning of the construct (`Mult` x y k) is *"multiply* `x` *with* `y`*, then send the result of this multiplication to the continuation* `k`*"*.

From Module additions.Addition_Chains

```
Inductive computation {A:Type}  : Type :=
| Return (a : A)
| Mult (x y : A) (k : A -> computation).
```

10.6.1.2 Monadic notation

The following *monadic* notation makes terms of type `computation` look like expressions of a small programming language dedicated to sequences of multiplications. Please look at *CPDT* [Chl11] for more details on monadic notations in Coq.

```
Notation "z '<---'  x 'times' y ';' e2 " :=
  (Mult x y  (fun z => e2))
    (right associativity, at level 60).
```

The `computation` type family is able to express sharing of intermediate computations. For instance, the computation of 2^7 depicted in Figure 10.3 is described by the following term:

```
Example comp7 : computation  :=
  x <--- 2 times 2;
  y <--- x times 2;
  z <--- y times y ;
  t <--- 2 times z ;
  Return t.
```

10.6.1.3 Definition

Thanks to the `computation` type family, we can associate a type to the kind of computation schemes described in Figures 10.1 and 10.2.

We define *addition chains* (in short *chains*) as functions that map any type `A` and any value `a` of type `A` into a computation on `A`:

```
Definition chain := forall A:Type, A -> @computation A.
```

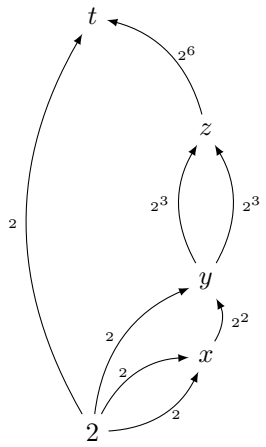


Figure 10.3: The dag associated to a computation of 2^7

```
Example  C87 : chain :=
 fun A (x : A) =>
  x2 <--- x times x ;
  x3 <--- x2 times x ;
  x6 <--- x3 times x3 ;
  x7 <--- x6 times x ;
  x10 <--- x7 times x3 ;
  x20 <--- x10 times x10 ;
  x40 <--- x20 times x20 ;
  x80 <--- x40 times x40 ;
  x87 <--- x80 times x7 ;
  Return x87.
```

Figure 10.4: A chain for raising x to its 87-th power

Thus, terms of type `chain` describe polymorphic exponentiation algorithms.

For instance, Fig 10.4 shows a definition of the chain of Figure 10.1, for the exponent 87. Note that, like in PHOAS, bound variables associated with the intermediary results are Coq variables of type A.

The structure of the definition of types `computation` and `chain` suggest that basic definitions over `chain` will have the following structure:

- A recursive function on type `computation` A (for a given type A)
- A main function on type `chain` that calls the previous one on any A`:Type`.

For instance, the following function computes the length of any chain, *i.e.*, the number of multiplications of the associated computation. Note that the function `chain_length` calls the auxiliary function `computation_length`, with the variable `A` instantiated to the singleton type `unit`.

Any other type in Coq would have fitted our needs, but `unit` and its unique inhabitant `tt` was the simplest solution.

```
Fixpoint computation_length {A} (a:A)(m : @computation A)
  : nat :=
match m with
  | Mult _ _ k => S (computation_length a (k a))
  | _ => 0%nat
end.

Definition chain_length (c:chain)
   := computation_length tt (c _ tt).

Compute chain_length C87.
```

```
= 9 : nat
```

10.6.2 Chains as a (small) programming language

The `chain` type can be considered as a tiny programming language dedicated to compute powers in any `EMonoid`. Thus, we have to define a semantics for this language. This semantics is defined in two parts:

- A structurally recursive function, — parameterized with an `EMonoid M` on a given type `A` —, that computes the value associated with any computation on `M`
- A polymorphic function that takes as arguments a chain `c`, a type `A`, an `EMonoid` on `A`, and a value `x:A`, then executes the computation `(c A x)`.

```
Fixpoint computation_execute  {A:Type} (op: Mult_op A)
                              (c : computation) :=
match c with
| Return x => x
| Mult x y k => computation_execute op (k (x * y))
end.

Definition chain_execute (c:chain) {A} op  (a:A) :=
  computation_execute op (c A a).
```

```
Definition computation_eval `{M:@EMonoid A E_op E_one E_eq}
           (c : computation) : A := computation_execute E_op c.

Definition chain_apply (c:chain)
    {M:@EMonoid A E_op E_one E_eq} a : A :=
    computation_eval (c A a).
```

Project 10.1 Study how to compile efficiently such data structures.

Examples:

The following interactions show how to apply the chain `C87` for exponentiation within two different monoids:

```
Compute  chain_apply C87 3%Z.
```

```
 =  3232579099291745342922739807213602718533873%Z
     : Z
```

```
Compute chain_apply C87 (M:=M2N) (Build_M2 1 1 1 0)%N.
```

```
  = {|
      c00 := 11000877783661019313%N;
      c01 := 6798916376386122583%N;
      c10 := 6798916376386122583%N;
      c11 := 4201961407274896733%N |}
     : M2 N
```

Project 10.2 Define a function which returns the sequence of operations defined by a chain. For instance, the chain `C87` of Figure 10.4 can be represented as a list containing terms of the form `(`i`, Add` j k`)` whenever the associated computation contains the operation $x^i = x^j \times x^k$.

```
Compute chain_trace C87.
```

```
(1, Init)
       :: (2, Add 1 1)
          :: (3, Add 2 1)
             :: (6, Add 3 3)
                :: (7, Add 6 1)
                   :: (10, Add 7 3)
                      :: (20, Add 10 10)
                         :: (40, Add 20 20)
                            :: (80, Add 40 40) :: (87, Add 80 7) :: nil
     : list (positive * info)
```

Note A first solution (in additions.Trace_exercise) consists in the definition of a (non-associative) multiplication over a type of trace, and apply the function `chain_execute` as if it were computing a power of `(1,Init)`.

10.6.2.1 Chain correctness and optimality

A chain is said to be *correct* with respect to a positive integer `p` if its execution in any monoid computes p-th powers.

```
Definition chain_correct_nat (n:nat) (c: chain) :=
  n <> 0 /\
  forall `(M:@EMonoid  A E_op E_one E_eq) (x:A),
      chain_apply c x ==   x ^ n.

Definition chain_correct (p:positive) (c: chain) :=
  chain_correct_nat c (Pos.
to_nat p).
```

Definition 10.1 *A chain c is* optimal *for a given exponent p if its length is less than or equal to the length of any chain correct for p.*

```
Definition optimal (p:positive) (c : chain) :=
 forall c', chain_correct p c' ->
            (chain_length c <= chain_length c')%nat.
```

10.7 Proving a chain's correctness

In this section, we present various ways of proving that a given chain is correct w.r.t. a given exponent. First, we just try to apply the definition in Section 10.6.2.1 on the preceding page, but this method is very inefficient, even for small exponents. In a second step, we use more sophisticated techniques such as reflection and parametricity. Automatic generation of correct chains will be treated in Sect. 10.8 on page 230.

10.7.1 Proof by rewriting

Let us show how to prove the correctness of some chains, using the `EMonoid` laws shown in Sect. 10.3.5 on page 205.

```
Ltac slow_chain_correct_tac :=
  match goal with
      [ |- chain_correct ?p ?c ] =>
      let A := fresh "A" in
      let op := fresh "op" in
      let one := fresh "one" in
      let eqv := fresh "eqv" in
      let M := fresh "M" in
      let x := fresh "x"
      in  split;
        [discriminate |
         unfold c, chain_apply, computation_eval; simpl;
         intros A op one eq M x; monoid_simpl M; reflexivity]
  end.


Example C7_ok : chain_correct 7 C7.
Proof.
   slow_chain_correct_tac.
Qed.
```

Unfortunately, this approach is terribly inefficient, even for quite small exponents:

```
Example C87_ok : chain_correct 87 C87.
Proof.
 Time  slow_chain_correct_tac.
```

```
Finished transaction in 62.808 secs (62.677u,0.085s) (successful)
```

```
Qed.
```

In addition to this big computation time, this approach generates a huge proof term. Just try to execute the command "`Print C87_ok`" to get a measure of its size. In order to understand this poor performance, let us consider an intermediate subgoal of the previous proof generated after a sequence of unfoldings and simplifications. This goal is presented below.

```
1 subgoal, subgoal 1 (ID 219)

  A : Type
  E_op : Mult_op A
  E_one : A
  E_eq : Equiv A
  M : EMonoid E_op E_one E_eq
  x : A
  ============================
   x * x * x * (x * x * x) * x * (x * x * x) *
   (x * x * x * (x * x * x) * x * (x * x * x)) *
   (x * x * x * (x * x * x) * x * (x * x * x) *
    (x * x * x * (x * x * x) * x * (x * x * x))) *
   (x * x * x * (x * x * x) * x * (x * x * x) *
    (x * x * x * (x * x * x) * x * (x * x * x)) *
    (x * x * x * (x * x * x) * x * (x * x * x) *
     (x * x * x * (x * x * x) * x * (x * x * x)))) *
   (x * x * x * (x * x * x) * x) ==
   x *
   (x *
    (x *
     (x *
      (x *
       (x *
        (x *
         (x *
          (x *
           (x *
            (x *
             (x * (x * (x * (x * (x * (x * (x * (x * (x * (x *
              ..))))))))))))))))))
```

This goal is solved by the following tactic composition:

```
monoid_simpl M; reflexivity.
```

This inefficiency certainly comes from the cost of setoid rewriting. At every application of an `EMonoid` law, the system must verify that the context of this rewriting is compatible with the equivalence relation associated with the current `EMonoid`. The rest of this chapter is devoted to the presentation of more efficient methods for proving chain correctness.

10.7.2 Correctness proofs by reflection

Instead of letting the tactic `rewrite` look for contexts in which setoid rewriting is possible, we propose to use (deterministic) computations for obtaining a "canonical" form for terms generated from a variable `x` by constructors associated with monoid multiplication and neutral element.

The reader will find general explanations about proofs by reflection in Coq, for instance in Chapter 16 of Coq'Art[BC04] and the numerous examples (including the `ring` tactic) in Coq's reference manual.

10.7.2.1 How does reflection work

Let us consider again the subgoal on page 221, the conclusion of which has the form $|a_1 == a_2|$, where $|a_1|$ and $|a_2|$ are terms of type `A`. Instead of spending space and time in setoid rewritings, we would like to normalize the terms $|a_1|$ and $|a_2|$ and verify that the associated normal forms are equal.

Defining such a normalization function is possible on an inductive type. The following type describes expressions composed of monoid operations and inhabitants of a given type A.

```
(** Binary trees of multiplications over A *)

Inductive Monoid_Exp (A:Type) : Type :=
 Mul_node (t t' : Monoid_Exp A) | One_node | A_node (a:A).

Arguments Mul_node {A} _ _.
Arguments One_node {A} .
Arguments A_node {A} _ .
```

Thus, the main steps of a correctness proof of a given chain, *e.g.*, `C87` will be the following ones:

1. generate a subgoal as in page 221,
2. express each term of the equivalence as the image of a term of type `Monoid_Exp` A,
3. normalize both terms and verify that their normal forms are equal.

The rest of this section is devoted to the definition of the normalization function on `Monoid_Exp` A, and the proofs of lemmas that link equivalence on type `A` and equality of normal forms of terms of type `Monoid_Exp` A.

10.7.2.2 Linearization function

The following functions help to transform any term of type `Monoid_Exp` A into a flat "normal form".

```
Fixpoint flatten_aux {A:Type} (t fin : Monoid_Exp A)
  : Monoid_Exp A :=
match t with Mul_node  t t' =>
              flatten_aux t (flatten_aux t' fin)
```

```
           | One_node  => fin
           |  x => Mul_node  x fin
end.

Fixpoint flatten {A:Type} (t: Monoid_Exp A) : Monoid_Exp A :=
match t with
| Mul_node t t' => flatten_aux t (flatten t')
| One_node => One_node
| X => Mul_node X One_node
end.
```

10.7.2.3 Interpretation function

The function `eval` maps any term of type `Monoid_Exp` A into a term of type A.

```
Function eval {A:Type} {op one eqv}
         (M: @EMonoid A op one eqv)
         (t: Monoid_Exp A) : A :=
 match t with
            | Mul_node t1 t2 => (eval M t1 * eval M t2)%M
            | One_node => one
            | A_node a => a
end.
```

The following two lemmas relate the linearization function `flatten` with the interpretation function `eval`.

```
Lemma flatten_valid {A} `(M: @EMonoid A op one eqv):
forall t , eval M t == eval M (flatten t).
(* Proof omitted *)

Lemma flatten_valid_2 {A} `(M: @EMonoid A op one eqv):
forall t t' , eval  M (flatten t) == eval M (flatten t')  ->
     eval M t == eval M t'.
(* Proof omitted *)
```

10.7.2.4 Transforming a multiplication into a tree

Let us now build a tool for building terms of type `Monoid_Exp` A out of terms of type `A` containing multiplications of the form `(_ * _)%M` and the variable `one`. In fact, what we want to define is an inverse of the function `flatten`.

Since `mult_op` is not a constructor (see Sect. 10.3.1), the transformation of a product of type `A` into a term of type `Monoid_Exp A` is done with the help of a tactic:

```
(** "Quote" tactic *)

Ltac model A  op one v :=
match v with
| (?x  * ?y)%M => let r1 := model A op one x
                  with r2 := model A op one y
```

```
                  in  constr:(@Mul_node A r1 r2)
| one => constr:(@One_node A)
| ?x => constr:(@A_node A x)
end.
```

For instance, the term `(x * x * x * (x * x * x) * x)` is transformed by `model` in the following term of type `Monoid_Exp` A

```
(eval M
   (Mul_node
     (Mul_node
        (Mul_node (Mul_node (A_node x) (A_node x)) (A_node x))
        (Mul_node (Mul_node (A_node x) (A_node x)) (A_node x)))
     (A_node x)))
```

10.7.3 Reflection tactic

The tactic `monoid_eq_A` converts a goal of the form (`E_eq` X Y), where X and Y are terms of type A, into (`E_eq (eval M (model X)) (eval M (model Y))`). This last goal is intended to be solved thanks to the lemma `flatten_valid_2`.

```
Ltac monoid_eq_A A op one E_eq M  :=
match goal with
| [ |- E_eq  ?X ?Y ] =>
  let tX := model A op one X with
      tY := model A op one Y in
      (change (E_eq (eval M tX) (eval M tY)))
end.
```

10.7.3.1 Main reflection tactic

The tactic `reflection_correct_tac` tries to prove a chain's correctness by a comparison of two terms of type `Monoid_Exp` A: one being obtained from the chain's definition, the other one by expansion of the naive exponentiation definition.

```
Ltac reflection_correct_tac :=
match goal with
[ |- chain_correct ?n ?c ] =>
 split; [try discriminate |
         let A := fresh "A"
         in let op := fresh "op"
         in let one := fresh "one"
         in let E_eq := fresh "eq"
         in let M := fresh "M"
         in let x := fresh "x"
         in  (try unfold c); unfold chain_apply;
           simpl; red; intros  A op one E_eq M x;
           unfold computation_eval;simpl;
           monoid_eq_A A op one E_eq M;
           apply flatten_valid_2;try reflexivity
```

```
        ]
end.
```

10.7.3.2 Example

The following dialogue clearly shows the efficiency gain over naive setoid rewriting.

```
Example C87_ok : chain_correct 87 C87.
Proof.
  Time reflection_correct_tac.
```

```
Finished transaction in 0.038 secs (0.038u,0.s) (successful)
```

```
Qed.
```

This tactic is not adapted to much bigger exponents. In Module Euclidean_Chains, for instance, we tried to apply this tactic for proving the correctness of a chain associated with the exponent 45319. We had to interrupt the prover, which was trying to build a linear tree of $2 \times 45319 + 1$ nodes! Indeed, using `reflection_correct_tac` is like doing a symbolic evaluation of an inefficient (linear) exponentiation algorithm.

In the next section, we present a solution that avoids doing such a lot of computations.

10.7.4 Chain correctness for —practically — free!

10.7.4.1 About parametricity

Let us now present another tactic for proving chain correctness, in the tradition of works on *parametricity* and its use for proving properties on programs. Strachey [Str00] explores the nature of *parametric polymorphism*: "*Polymorphic functions behave uniformly for all types*" then Reynolds [Rey83] formalizes this notion through binary relations. Wadler [Wad89], then Cohen *et al.* [CDM13] use this relation for deriving theorems about functions that operate on parametric polymorphic types.

Let us look again at the definitions of type family `computation` and the type `chain`:

```
Inductive computation {A:Type}  : Type :=
| Return (a : A)
| Mult (x y : A) (k : A -> computation).

Definition chain := forall A:Type, A -> @computation A.
```

Let c be a closed term of type `chain`; c is of the form `fun (A:Type)(a:A) =>` t_a, where t_a is a term of type `@computation A`. Obviously, in every subterm of t_a of type `A`, the two first arguments of constructor `Mult` or the argument of `Return` are either `a` or a variable introduced as the

formal argument of a continuation `k`. In effect, there is no other way to build terms of type `A` in the considered context.

Marc Lasson's **paramcoq** plug-in (available as `opam` package `coq-paramcoq`) generates a family of binary relations definitions from `computation`'s definition.

```
Inductive
computation_R (A B : Type) (R : A -> B -> Type)
  : computation -> computation -> Type :=
 |    computation_R_Return_R :
       forall (a1 : A) (a2 : B),  R a1 a2 ->
         computation_R A B R (Return a1) (Return a2)
  | computation_R_Mult_R : forall (x1 : A) (x2 : B),
         R x1 x2 ->
          forall (y1 : A) (y2 : B),
            R y1 y2 ->
            forall (k1 : A -> computation)
                   (k2 : B -> computation),
                      (forall (H : A) (H0 : B),
                          R H H0 ->
                          computation_R A B R (k1 H) (k2 H0)) ->
                       computation_R A  B R
                          (z <--- x1 times y1; k1 z)
                          (z <--- x2 times y2; k2 z)
```

Let A and B be two types, and $R : A \to B \to \mathbf{Type}$ a relation. Two computations `cA: @computation A` and `cB: @computation B` are related *w.r.t.* `computation_R` if every pair of arguments of `Mult` and `Return` at the same position are related *w.r.t.* `R`.

10.7.4.2 Definition

A chain c is *parametric* if it has the same behavior for any pair of types A and B, any relation R between A and B and any R-related pair of arguments a and b:

```
Definition parametric (c:chain) :=
  forall A B (R: A -> B -> Type) (a:A) (b:B),
   R a b -> computation_R  R (c A a) (c B b).
```

10.7.4.3 How to use these definitions?

Let us use parametricity for proving easily a given chain's correctness. In other words, let c be a chain and p`:positive` be a given exponent. Consider some instance of `EMonoid` over a type A. We want to prove that the application of the chain c to any value a of type A returns the value a^p.

We first use Coq's computation facilities for "guessing" the exponent associated with any given chain. It suffices to instantiate "monoid multiplication" with addition on positive integers.

```
Definition the_exponent_nat (c:chain) : nat :=
 chain_apply c (M:=Natplus) 1%nat.
```

```
Definition the_exponent (c:chain) : positive :=
  chain_execute c Pos.add  1%positive.

Compute the_exponent C87.
```

```
= 87%positive
    : positive
```

We show how to *prove* that a given chain c, applied to any a, really computes a^p, where $p =$ the_exponent c. Parametricity allows us to compare executions on any monoid M with executions on `NatPlus`. Let us consider the following mathematical relation

$$\{(x, n) \in M \times \mathbb{N} \,|\, 0 < n \wedge x = a^n\}$$

```
Definition power_R  (a:A) :=
  fun (x:A)(n:nat) => n <> 0 /\ x == a ^ n.
```

First, we prove the following lemma, that relates `computation_R` with the result of the executions of the corresponding computations:

```
Lemma  power_R_is_a_refinemnt (a:A) :
  forall(gamma : @computation A)
        (gamma_nat : @computation nat),
    computation_R  (power_R a) gamma gamma_nat ->
     power_R a (computation_eval gamma)
               (computation_eval (M:= Natplus) gamma_nat).
(* Proof omitted *)
```

Thus, if c:`chain` is parametric, this refinement lemma allows us to prove a correctness result:

```
Lemma param_correctness_nat :
 forall c:chain, parametric c ->
               chain_correct_nat (the_exponent_nat c)  c.
(* Proof omitted *)
```

A similar result can be proven with the exponent in `positive`. First we instantiate the parameter `R` of `computation_R`, with the relation that links the representations of natural numbers on respective types `nat` and `positive`. Then we use our lemmas for rewriting under the assumption that the considered chain is parametric. Please note how our approach is related with *data refinement* (see also [CDM13]). The reader may also consult a survey by D. Brown on the most important contributions to the notion of parametricity [Bro10].

```
Lemma exponent_pos2nat : forall c: chain,  parametric c ->
  the_exponent_nat c = Pos.to_nat (the_exponent c).

Lemma exponent_pos_of_nat : forall c: chain,  parametric c ->
  the_exponent c = Pos.of_nat (the_exponent_nat c).
```

```
Lemma param_correctness (c:chain) :
   parametric c ->
   chain_correct (the_exponent c) c.
Proof.
  intros; rewrite  exponent_pos_of_nat; auto.
  red;  rewrite  exponent_pos2nat;auto.
  rewrite Pos2Nat.id,  <- exponent_pos2nat;auto.
  apply param_correctness_nat; auto.
Qed.
```

Lemma `param_correctness` suggests us a method for verifying that a given chain c is correct *w.r.t.* some positive exponent p:

1. Verify that c is parametric.
2. Verify that p is equal to (`the_exponent` c).

10.7.4.4 How to prove a chain's parametricity

Despite the apparent complexity of `computation_R`'s definition, it is very simple to prove that a given chain is parametric. The following tactics proceed as follows:

1. Given a chain c, consider two types `A` and `B`, and any relation `R:A->B->Prop`,
2. Push into the context declarations of `a:A`, `b:B` and an hypothesis assuming `R a b`.
3. Then the tactic crosses in parallel the terms (`c A a`) and (`c B b`) (of the same structure),
   - On a pair of terms of the form `Mult xA yA (fun zA => tA)` and `Mult xB yB (fun zB => tB)`, the tactic checks whether `R xA xB` and `R yA yB` are already assumed in the context, then pushes into the context the declaration of `zA` and `zB` and the hypothesis `Hz: R zA zB`, then crosses the terms `tA` and `tB`
   - On a pair of terms of the form (`Return xA`) and (`Return xB`), the tactic just checks whether (`R xA xB`) is assumed.

The tactic itself is simpler than its explanation.

```
Ltac parametric_tac  :=
match goal with [ |- parametric ?c] =>
   red ; intros;
   repeat (right;[assumption | assumption | ]);
   left; assumption
end.

Example P87 : parametric C87.
Proof. Time parametric_tac.
```

```
Finished transaction in 0.005 secs (0.005u,0.s) (successful)
```

```
Qed.
```

10.7.4.5 Proving a chain's correctness

Finally, for proving that a given chain c is correct with respect to an exponent p, it suffices to check that c is parametric, and to apply the lemma `param_correctness`. The reader will note how this computation-less method is much more efficient than our reflection tactic.

```
Ltac param_chain_correct :=
match goal with
[|- chain_correct ?p ?c ] =>
apply param_correctness; parametric_tac
end.

Lemma C87_ok' : chain_correct 87 C87.
Time param_chain_correct.
```

```
Finished transaction in 0.005 secs (0.005u,0.s) (successful)
```

```
Qed.
```

10.7.4.6 Remark

For the reasons exposed in Section 10.7.4.1 on page 225, it seems obvious that any well-written chain is parametric. Unfortunately, we cannot prove this property in Coq, for instance by induction on `c`, since `chain` is a product type and not an inductive type.

```
Definition any_chain_parametric : Type :=
 forall c:chain, parametric c.

Goal any_chain_parametric.
Proof.
intros c A B R a b ; induction c.
```

```
 2 subgoals, subgoal 1 (ID 556)

  c : chain
  A : Type
  B : Type
  R : A -> B -> Type
  a : A
  b : B
  a0 : A
  ============================
   R a b -> computation_R R (Return a0) (c B b)

...
```

```
Abort.
```

Given this situation, we could admit (as an axiom) that any chain is parametric. Nevertheless, if a chain is under the form of a closed term, using `parametric_tac` is so efficient than we prefer to avoid a shameful introduction of an axiom in our development.

10.8 Certified chain generators

In this section, we are interested in the *correct by construction* paradigm. We just want to give a positive exponent to Coq and get a (hopefully) correct and efficient chain for this exponent.

We first define the notion of *chain generator*, then present a certified generator that simulates the binary exponentiation algorithm. Last, we present a better chain generator based on integer division.

10.8.1 Definitions

We call *chain generator* any function that takes as argument any positive integer and returns a chain.

```
Definition chain_generator := positive -> chain.
```

A generator g is *correct* it it returns a correct chain for any exponent:

```
Definition correct_generator (g : positive -> chain) :=
 forall p, chain_correct (g p) p.
```

Correct generators can be used for computing powers on the fly, thanks to the following functions:

```
Definition cpower_pos (g : chain_generator)  p
           `{M:@EMonoid A E_op E_one E_eq} a :=
  chain_apply (g p) (M:=M) a.


Definition cpower (g : chain_generator)  n
           `{M:@EMonoid A E_op E_one E_eq} a :=
  match n with 0%N => E_one
             | Npos p => cpower_pos  g p a
  end.
```

Note also that the use of chain generators is independent from the techniques presented in Sect. 10.7: Designing an efficient and correct chain generator may be a long and hard task. On the other hand, once a generator is certified, we are assured of the correctness of all its outputs. Finally, we say that a generator g is *optimal* if it returns chains whose length are less than or equal to any chain returned by any correct generator:

```
Definition optimal_generator (g : positive -> chain) :=
 forall p:positive, optimal p (g p).
```

10.8.2 The binary chain generator

Let us reinterpret the binary exponentiation algorithms in the framework of addition chains. Instead of directly computing x^n for some base x and exponent n, we build chains that describe the computations associated with the binary exponentiation method. Not surprisingly, this chain generation will be described in terms of recursive functions, once the underlying monoid is fixed.

As for the "classical" binary exponentiation algorithm, we define an auxiliary computation generator for the product of an accumulator a with an arbitrary power of some value x. Then, the main function builds a computation for any positive exponent:

```
Fixpoint axp_scheme  {A} p : A -> A -> @computation A   :=
 match p with
   | xH =>  (fun a x => y <--- a  times x ; Return y)
   | xO q => (fun a x => x2 <--- x times  x ; axp_scheme q a x2)
   | xI q => (fun a x => ax <--- a times x ;
                         x2 <--- x times x ;
                         axp_scheme q ax x2)
end.

Fixpoint  bin_pow_scheme {A} (p:positive)
: A -> @computation A:=
  match p with
  |  xH => fun x => Return x
  | xI q  => fun x => x2 <--- x times x; axp_scheme q x x2
  | xO q => fun x => x2 <--- x times x ; bin_pow_scheme q x2
  end.
```

The following function associates a chain to any positive exponent:

```
Definition binary_chain (p:positive) : chain :=
  fun A => bin_pow_scheme p.

Compute binary_chain 87.
```

```
    = fun (A : Type) (x : A) =>
      x0 <--- x times x;
      x1 <--- x times x0;
      x2 <--- x0 times x0;
      x3 <--- x1 times x2;
      x4 <--- x2 times x2;
      x5 <--- x4 times x4;
      x6 <--- x3 times x5;
      x7 <--- x5 times x5;
      x8 <--- x7 times x7;
```

```
      x9 <--- x6 times x8;
      Return x9
    : chain
```

10.8.2.1 Proof of `binary_chain`'s correctness

Let us now prove that `binary_chain` always returns correct chains. First, due to the structure of this generator's definition, we study the properties of the auxiliary functions that operate *on a given monoid* M.

```
Section binary_power_proof.

Variables (A: Type)
         (E_op : Mult_op A)
         (E_one : A)
         (E_eq: Equiv A).

Context (M : EMonoid  E_op E_one E_eq).

Existing Instance Eop_proper.
```

```
Lemma axp_correct : forall p a x,
 computation_eval (axp_scheme p a x) == a *  x ^ (Pos.to_nat p).
(*  Proof by induction on p *)

Lemma binary_correct :
  forall p x,
        computation_eval  (bin_pow_scheme p (A:=A) x) ==
        x ^ (Pos.to_nat p).
(* Proof by induction on p *)

End  binary_power_proof.
```

```
Lemma binary_generator_correct : correct_generator binary_chain.
Proof.
  red;unfold chain_correct,  binary_chain, chain_apply;
  split; [auto| intros A op one Eq M x; apply binary_correct].
Qed.
```

10.8.2.2 The binary method is not optimal

It is easy to prove by contradiction that the binary method is not the most efficient for computing powers. First, let us assume that `binary_chain` is optimal:

```
Section non_optimality_proof.

 Hypothesis binary_opt : optimal binary_chain.
```

Then, let us consider for instance the binary chain generated for the exponent 87.

```
Compute chain_length (binary_chain 87).
```

```
   = 10 : nat
```

Let us recall that `C87`'s length has been evaluated to 9 (Sect 10.6.1.3, and that this chain is correct (Sect 10.7.4.5 on page 229). Thus, it is very easy to finish our proof:

```
Lemma binary_generator_not_optimal : False.
Proof.
  generalize (binary_opt gen _  _ C87_ok);
  compute; omega.
Qed.

End non_optimality_proof.
```

Exercise 10.3 Prove that for any positive integer p, the length of any optimal chain for p is less than twice the number of digits of the binary representation of p.

10.9 Euclidean Chains

In this section, we present an efficient chain generator. The chains built by this generator are never longer than the chains built by the binary generator. Moreover, for an infinite number of exponents, the chains it builds are strictly shorter than the chain returned by `binary_chain`. Euclidean chains are based on the following idea:

> For generating a chain that computes x^n, one may choose some natural number $0 < p < n$, and build a chain that computes first x^p **then** uses this value for computing x^n.

For instance, a computation of x^{42} can be decomposed into a computation of $y = x^3$, then a computation of y^{14}. The efficiency of the chain built with this methods depends heavily on the choice of p. See [BCHM95] for details.

Considering chain generators and their correctness, we may consider the dual of decomposition of exponents: we would like to write *composable* correct chain generators. For instance, we want to build some object that, "composed" with any correct chain for n, returns a correct chain for $3n$.

10.9.0.0.1 Note: All the Coq material described in this section is available on Module additions/Euclidean_Chains.v

10.9.1 Chains and continuations : f-chains

Please consider the following small example:

```
Example C3 : chain :=
 fun A (x:A) =>
  x2 <--- x times x;
  x3 <--- x2 times x ;
  Return x3.
```

The execution of this chain on some value $x : A$ stops after computing x^3, because of the `Return` "statement". However, we would like to compose the instructions of `C3` with a chain for another exponent n, in order to generate a chain for the exponent $3 \times n$.

The solution we present is based on functional programming and the concept of continuation.

10.9.1.1 Type definition of f-chains

Let us consider *incomplete* or *open* chains. Such an object waits for another chain to resume a computation.

Figure 10.5 represents an f-chain associated with the exponent 3, as a dag with an input and one output the edges of which are depicted as thick arrows.

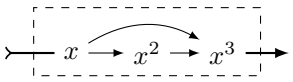


Figure 10.5: Graphical representation of `F3`

In other words, this kind of objects can be considered as *functions* from chains to chains. So, we called their type `Fchain`.

First, we define a type of *continuations*, *i.e.*, functions that wait for some value x, then build a computation for raising x to some given exponent.

```
Definition Fkont (A:Type) := A -> @computation A.
```

An `f-chain` is just a polymorphic function that combines a continuation and en element into a computation:

```
Definition Fchain := forall A, Fkont A -> A -> @computation A.
```

10.9.1.2 Examples

Let us define a chain for computing the cube of some x, then sending the result to a continuation k.

```
Definition F3 : Fchain :=
 fun  A k  (x:A) =>
  y <--- x times x ;
  z <--- y times x ;
  k  z.
```

Any f-chain can be converted into a chain by the help of the following function:

```
Definition F2C (f : Fchain) : chain :=
 fun (A:Type) => f A Return.

Compute the_exponent (F2C F3).
```

```
= 3%nat
```

In the rest of this chapter, we will use two other f-chains, respectively associated with the exponents 1 and 2. Chains `F1`, `F2` and `F3` will form a basis to generate chains for many exponents by *composition of correct functions*.

```
Definition F1 : Fchain :=
 fun A k (x:A) => k x.

Definition F2 : Fchain :=
fun  A k  (x:A) =>
  y <--- x times x ;
  k  y.
```

10.9.1.3 F-chain application and composition

The following definition allows us to consider any value f of type `Fchain` as a function of type `chain` $\rightarrow$ `chain`.

```
Definition Fapply (f : Fchain) (c: chain) : chain  :=
 fun A  x =>  f  A (fun y => c A  y) x.
```

In a similar way, *composition* of `f-chain`s is easily defined (see Figure 10.6).

```
Definition Fcompose (f1 f2: Fchain) : Fchain  :=
 fun   A k x =>  f1  A (fun y => f2 A k y) x.

Lemma F1_neutral_l : forall f, Fcompose F1 f = f.
Proof. reflexivity. Qed.

Lemma F1_neutral_r : forall f, Fcompose f F1 = f.
Proof. reflexivity. Qed.
```

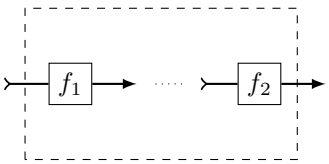


Figure 10.6: Composition of f-chains f_1 and f_2 (`Fcompose`)

10.9.1.4 Examples

The following examples show that the apparent complexity of the previous definition is counterbalanced with the simplicity of using `Fapply` and `Fcompose`.

```
Example F9 := Fcompose F3 F3.

Compute F9.
```

```
= fun (A : Type) (x : Fkont A) (x0 : A) =>
      x1 <--- x0 times x0;
      x2 <--- x1 times x0; x3 <--- x2 times x2;
      x4 <--- x3 times x2;
      x x4
    : Fchain
```

```
Remark F9_correct :chain_correct 9 (F2C F9).
Proof.
  apply param_correctness_pos;  lazy;  parametric_tac.
Qed.
```

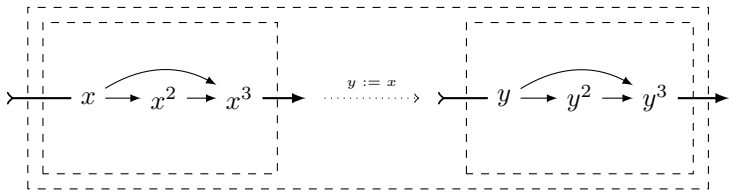


Figure 10.7: Composition of F-chains: F9

Using structural recursion and the operator `FCompose`, we build a chain for any exponent of the form 2^n:

```
Fixpoint  Fexp2_of_nat (n:nat) : Fchain :=
match n with O => F1
           | S p => Fcompose F2 (Fexp2_of_nat p)
end.


Definition Fexp2 (p:positive) : Fchain :=
  Fexp2_of_nat (Pos.to_nat p).

Compute Fexp2  4.
```

```
= fun (A : Type) (x : Fkont A) (x0 : A) =>
      x1 <--- x0 times x0;
      x2 <--- x1 times x1; x3 <--- x2 times x2;
      x4 <--- x3 times x3; x x4
    : Fchain
```

10.9.2 F-chain correctness

Let `f` be some term of type `Fchain`, and `n:nat`. We would like to say that `f` is correct *w.r.t.* `n:nat` if for any continuation `k` and `a`, the application of `f` to `k` and `a` computes $k(a^n)$.

```
Module Bad.

Definition Fchain_correct  (n:nat) (f : Fchain) :=
  forall A `(M : @EMonoid A op E_one E_equiv) k (a:A),
    computation_execute op (f A k  a)==
    computation_execute op (k  (a ^ n)).
```

Let us now try to prove that `F3` is correct *w.r.t.* 3.

```
Theorem F3_correct : Fchain_correct 3 F3.
Proof.
  intros    A op E_one E_equiv M k  a ; simpl.
  monoid_simpl M.
```

```
  A : Type
  op : Mult_op A
  E_one : A
  E_equiv : Equiv A
  M : EMonoid op E_one E_equiv
  k : Fkont A
  a : A
  H : Proper (equiv ==> equiv ==> equiv) op
  ============================
   computation_execute op (k (a * a * a)) ==
   computation_execute op (k (a * (a * (a * E_one))))
```

```
Abort.
End Bad.
```

This failure is due to a lack of an assumption that the continuation `k` is *proper* with respect to the equivalence `equiv`. Thus, Coq is unable to infer from the equivalence `(a * a * a) == (a * (a * (a * E_one)))` that `k (a * a * a)` and `k (a * (a * (a * E_one)))` are equivalent computations.

10.9.2.1 Definition:

A continuation `k:Fkont A` is *proper* if, whenever `x == y` holds, the computations `k x` and `k y` are equivalent.

```
Class Fkont_proper
      `(M : @EMonoid A op E_one E_equiv) (k: Fkont A )  :=
  Fkont_proper_prf:
    Proper (equiv ==> computation_equiv op E_equiv) k.
```

We are now able to improve our definition of correctness, taking only proper continuations into account.

```
Definition Fchain_correct_nat (n:nat) (f : Fchain) :=
 forall A `(M : @EMonoid A op E_one E_equiv) k
        (Hk :Fkont_proper M k)
        (a : A) ,
 computation_execute op (f A k  a) ==
 computation_execute op (k  (a ^ n)).


Definition Fchain_correct (p:positive) (f : Fchain) :=
 Fchain_correct_nat (Pos.to_nat p) f.
```

10.9.2.2 Examples

Let us show some manual correctness proofs for small f-chains:

```
Lemma F1_correct : Fchain_correct 1 F1.
Proof.
  intros until M ; intros k Hk a ; unfold F1; simpl.
  apply Hk; monoid_simpl M; reflexivity.
Qed.
```

While proving F3's correctness, we will have to apply the properness hypothesis on k:

```
Theorem F3_correct : Fchain_correct 3 F3.
Proof.
  intros until M; intros k Hk a; simpl.
```

```
 A : Type
  op : Mult_op A
  E_one : A
  E_equiv : Equiv A
  M : EMonoid op E_one E_equiv
  k : Fkont A
  Hk : Fkont_proper M k
  a : A
  ============================
   computation_execute op (k (a * a * a)) ==
   computation_execute op (k (a * (a * (a * E_one))))}
```

```
apply Hk.
```

```
...
============================
a * a * a == a * (a * (a * E_one))}
```

```
  monoid_simpl M; reflexivity.
Qed.
```

Correctness of F2 is proved the same way:

```
Theorem F2_correct : Fchain_correct 2 F2.
Proof.
  intros until M; intros k Hk a; simpl;
  apply Hk;  monoid_simpl M;  reflexivity.
Qed.
```

10.9.2.3 Composition of correct f-chains: a first attempt

We are now looking for a way to generate correct chains for any positive number. It seems obvious that we could use Fcompose for building a correct f-chain for $n \times p$ by composition of a correct f-chain for n and a correct f-chain for p.

Let us try to certify this construction:

```
Module Bad2.

Lemma Fcompose_correct_attempt :
  forall f1 f2 n1 n2, Fchain_correct n1 f1 ->
                      Fchain_correct n2 f2 ->
                      Fchain_correct (n1 * n2) (Fcompose f1 f2).

(* Beginning of proof omitted *)
```

```
  Hk : Fkont_proper M k
  a, x, y : A
  Hxy : x == y
  ============================
   computation_execute op (f2 A k x) ==
   computation_execute op (f2 A k y)
```

No hypothesis guarantees us that the execution of f2 respects the equivalence x == y.

```
Abort.
```

Thus, we need to define also a notion of properness for f-chains. A first attempt would be :

```
Module Bad3.

Class Fchain_proper_ (fc : Fchain) := Fchain_proper_prf :
 forall  `(M : @EMonoid A op E_one E_equiv) k  ,
    Fkont_proper M k
    forall x y, x == y ->
        @computation_equiv _ op E_equiv (fc A k x) (fc A k y).
```

This definition is powerful enough for proving that properness is preserved by composition:

```
Instance Fcompose_proper_ (f1 f2 : Fchain)
                            (_ : Fchain_proper_simple f1)
                            (_ : Fchain_proper_simple f2) :
 Fchain_proper_ (Fcompose f1 f2).
Proof.
 intros until M;intros k Hk x y Hxy; unfold Fcompose;cbn.
 apply (H _ _ _ _ M); auto.
 intros u v Huv;apply (H0 _ _ _ _ M);auto.
Qed.
```

Nevertheless, we had to throw away this definition of properness: In further developments (Sect. 10.9.3 on page 242) we shall have to compare executions of the form `fc A` k_x `x` and `fc A` k_y `y` where `x == y` and k_x and k_y are "equivalent" but not *convertible* continuations.

```
End Bad3.
```

10.9.2.4 A better definition of properness

The following generalization will allow us to consider continuations that are different (according to Leibniz equality) but lead to equivalent computations and results.

```
Definition Fkont_equiv  `(M : @EMonoid A op E_one E_equiv)
 (k k': Fkont A )  :=
 forall x y : A, x == y ->
                 computation_equiv op E_equiv  (k x)  (k' y).

Class Fchain_proper (fc : Fchain) := Fchain_proper_prf :
 forall  `(M : @EMonoid A op E_one E_equiv) k k' ,
    Fkont_proper M k -> Fkont_proper M k' ->
    Fkont_equiv M k k' ->
    forall x y, x == y ->
        @computation_equiv _ op E_equiv
                           (fc A k x)
                           (fc A k' y).
```

10.9.2.5 Examples

The definition above allows us to build simply several instances of the class `Fchain_proper`:

```
Instance F1_proper : Fchain_proper F1.
Proof.
  intros until M ; intros k k' Hk Hk' H a b H0; unfold F1; cbn;
  now apply H.
Qed.
```

```
Ltac add_op_proper M H :=
 let h := fresh H in
```

```
   generalize (@Eop_proper _ _ _ _ M); intro h.


Instance F3_proper : Fchain_proper F3.
Proof.
  intros  A op one equiv M  k k' Hk Hk'  Hkk' x y Hxy;
  apply Hkk'; add_op_proper M H; repeat rewrite Hxy;
  reflexivity.
Qed.
```

We are now able to prove `Fexp2` n's correctness by induction on n:

```
Instance Fexp2_nat_proper (n:nat) :
                          Fchain_proper (Fexp2_of_nat n).
Proof.
 induction n; cbn.
   - apply F1_proper.
   - apply Fcompose_proper ; [apply F2_proper | apply IHn].
Qed.
```

```
Lemma  Fexp2_nat_correct (n:nat) :
         Fchain_correct_nat (2  ^ n) (Fexp2_of_nat n).
Proof.
  induction n; cbn.
 - apply F1_correct.
 -  rewrite Nat.add_0_r;
   replace (2 ^ n + 2 ^ n)%nat with (2 * 2 ^n)%nat by  omega;
   apply Fcompose_correct_nat;auto.
   +  apply F2_correct.
   +  apply  Fexp2_nat_proper.
Qed.
```

```
Lemma  Fexp2_correct (p:positive) :
                          Fchain_correct (2  ^ p) (Fexp2 p).
(* Proof omitted *)

Instance  Fexp2_proper (p:positive) : Fchain_proper (Fexp2 p).
(* Proof omitted *)
```

We are now able to build chains for any exponent of the form $2^k \times 3^p$, using `Fcompose`. Les us look at a simple example:

```
Hint Resolve F1_correct F1_proper
     F3_correct F3_proper Fcompose_correct Fcompose_proper
     Fexp2_correct Fexp2_proper .

Example F144:  {f : Fchain | Fchain_correct 144 f /\
                                Fchain_proper f}.
Proof.
 change 144 with ( (3 * 3) * (2 ^ 4))%positive.
 exists (Fcompose (Fcompose F3 F3) (Fexp2 4)); auto.
```

```
Defined.


Compute proj1_sig F144.
```

```
= fun (A : Type) (x : Fkont A) (x0 : A) =>
      x1 <--- x0 times x0;
      x2 <--- x1 times x0;
      x3 <--- x2 times x2;
      x4 <--- x3 times x2;
      x5 <--- x4 times x4;
      x6 <--- x5 times x5;
      x7 <--- x6 times x6;
      x8 <--- x7 times x7;
      x x8
    : Fchain
```

10.9.3 Building chains for two distinct exponents : k-chains

10.9.3.1 Introduction

Not every chain can be built efficiently with `Fcompose`. For instance, consider the exponent $n = 23 = 3 + 2^4 + 2^2$.

One may attempt to define a new operator for combining f-chains for n and p into an f-chain for $n + p$.

```
Definition Fplus (f1 f2 : Fchain) : Fchain :=
  fun A k x =>
   f1 A (fun y =>
           f2 A (fun z => t <--- z times y; k t) x)
           x.
```

For instance, we can define a chain for 23:

```
Let F23 := Fplus F3 (Fplus (Fexp2 4) (Fexp2 2)).
```

Unfortunately, our construct is still very inefficient, since it results in duplication of computations, as shown by the normal form of `F23`.

```
Compute F23
```

```
 = fun (A : Type) (k : Fkont A) (x0 : A) =>
      x1 <--- x0 times x0;
      x2 <--- x1 times x0;
      x3 <--- x0 times x0;
      x4 <--- x3 times x3;
      x5 <--- x4 times x4;
      x6 <--- x5 times x5;
      x7 <--- x0 times x0;
      x8 <--- x7 times x7;
```

```
      x9 <--- x8 times x6;
      x10 <--- x9 times x2;
      k x10
```

We observe that the variables `x3` and `x7` are useless, since they will have the same value as `x1`. Likewise, computing `x8` (same value as `x4`) is a waste of time.

A better scheme for computing x^{23} would be the following one:

1. Compute x, x^2, x^3, **and** $x^6 = (x^3)^2$, then x^7,
2. Compute $x^{10} = x^7 \times x^3$, then x^{20}
3. Finally, return $x^{23} = x^{20} \times x^3$

In fact, the first step of this sequence computes *two* values: x^7 and x^3, that are re-used by the rest of the computation.

Like in some programming languages that allow "multiple values", like `Scheme` and `Common Lisp`, we can express this feature in terms of continuations that accept two arguments. Thus, we extend our previous definitions to chains that return two different powers of their argument[4].

```
Definition Kkont A:=  A -> A -> @computation A.

Definition Kchain :=  forall A, Kkont A -> A -> @computation A.
```

10.9.3.2 Examples

The chain `k3_1` sends both values x and x^3 to its continuation. Likewise, `k7_3` "returns" x^7 and x^3.

```
Example k3_1 : Kchain := fun A (k:Kkont A) (x:A) =>
  x2 <--- x times x ;
  x3 <--- x2 times x ;
  k x3 x.

Example k7_3 : Kchain := fun A (k:Kkont A)  (x:A) =>
  x2 <--- x times x;
  x3 <--- x2 times x ;
  x6 <--- x3 times x3 ;
  x7 <--- x6 times x ;
  k  x7 x3.
```

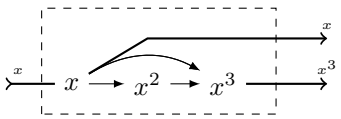


Figure 10.8: Graphical representation of `K3_1`

[4]The name `Kchain` comes from previous versions of this development. It may be changed later.

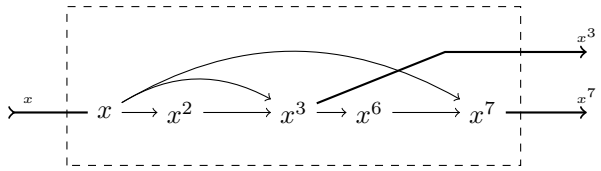


Figure 10.9: Graphical representation of `K7_3`

10.9.3.3 Definitions

First, we have to adapt to k-chains our definitions of correctness and properness.

```
Definition Kkont_proper `(M : @EMonoid A op E_one E_equiv)
           (k : Kkont A) :=
 Proper (equiv ==> equiv ==> computation_equiv op E_equiv) k .

Definition Kkont_equiv  `(M : @EMonoid A op E_one E_equiv)
           (k k': Kkont A )  :=
 forall x y : A, x == y -> forall z t, z == t ->
         computation_equiv op E_equiv   (k  x z) (k' y t).
```

A k-chain is correct with respect to two exponents n and p if it computes x^n and x^p for any x in any monoid M.

```
Definition Kchain_correct_nat (n p : nat) (kc : Kchain) :=
  forall `(M : @EMonoid A op E_one E_equiv)
        (k : Kkont A),
        Kkont_proper M k ->
        forall  (x : A) ,
              computation_execute op (kc  A k  x) ==
              computation_execute op (k  (x ^ n) (x ^ p)).
```

```
Definition Kchain_correct (n p : positive) (kc : Kchain) :=
  Kchain_correct_nat  (Pos.to_nat n) (Pos.to_nat p) kc.

Class Kchain_proper (kc : Kchain) :=
Kchain_proper_prf :
 forall `(M : @EMonoid A op E_one E_equiv) k k' x y ,
   Kkont_proper M k ->
   Kkont_proper M k' ->
   Kkont_equiv M k k' ->
   E_equiv x y ->
   computation_equiv op E_equiv (kc A k x) (kc A k' y).
```

10.9.3.4 Example

For instance, let us prove that `k7_3` is proper and correct for the exponents 7 and 3.

```
Instance k7_3_proper : Kchain_proper k7_3.
Proof.
```

```
  intros until M; intros; red; unfold k7_3; cbn;
  add_op_proper M H3; apply H1;  rewrite H2;   reflexivity.
Qed.

Lemma k7_3_correct : Kchain_correct 7 3 k7_3.
Proof.
  intros until M; intros; red; unfold k7_3; simpl.
  apply H; monoid_simpl M;  reflexivity.
Qed.
```

10.9.4 Systematic construction of correct f-chains and k-chains

We are now ready to define various operators on f- and k-chains, and prove these operators preserve correctness and properness. We will also show that these operators allow to generate easily correct chains for any positive exponent. They will be used to generate chains for numbers of the form $n = bq + r$ where $0 \leq r < b$, assuming the previous construction of correct chains for r, b and q. For instance, Figure 10.10 shows how `K7_3` is built as a composition of `K3_1` and `F2`.

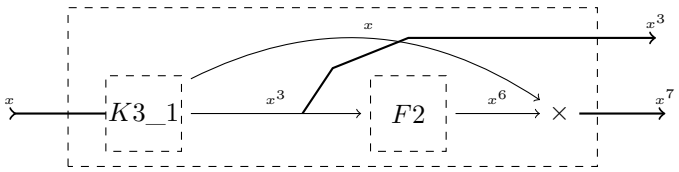


Figure 10.10: Decomposition of `K7_3`

10.9.4.1 Conversion from k-chains into f-chains

Any k-chain for n and p can be converted into an f-chain, just by applying it to a continuation that ignores its second argument.

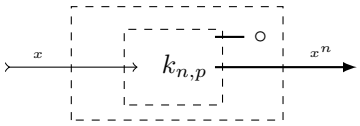


Figure 10.11: The `K2F (knp)` construction

```
Definition K2F (knp : Kchain) : Fchain :=
  fun A (k:Fkont A) => kc A (fun  y _ => k y).

Lemma K2F_correct :
  forall knp n p, Kchain_correct kc n p ->
                  Fchain_correct (K2F n) knp.
(* Proof omitted *)

Instance K2F_proper (kc : Kchain)(_ : Kchain_proper kc) :
```

```
                                    Fchain_proper (K2F kc).

(* Proof omitted} *)
```

10.9.4.2 Construction associated with Euclidean division with a positive rest

Let $n = bq + r$, with $0 < r < b$. Then, for any x, $x^n = (x^b)^q \times x^r$. Thus, we can compose an chain that computes x^b and x^r with a chain that raises any y to its q-th power for obtaining a chain that computes x^n.

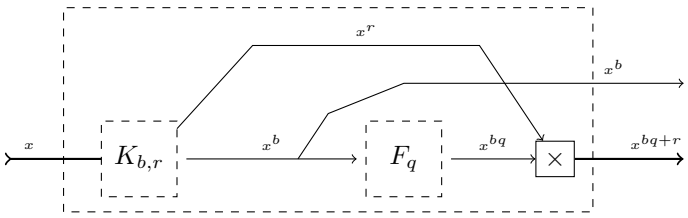


Figure 10.12: The KFK combinator

```
Definition KFK (kbr : Kchain) (fq : Fchain) : Kchain  :=
  fun A k a =>
    kbr A  (fun xb xr =>
              fq A (fun y =>
                      z <--- y times xr; k z xb) xb) a.

Lemma KFK_correct :
  forall (b q r : positive) (kbr : Kchain) (fq : Fchain),
    Kchain_correct b r  kbr ->
    Fchain_correct q fq ->
    Kchain_proper kbr ->
    Fchain_proper fq ->
    Kchain_correct  (b * q + r) b (KFK kbr fq).
(* Proof omitted *)

Instance KFK_proper :
forall (kbr : Kchain) (fq : Fchain),
  Kchain_proper kbr ->
  Fchain_proper fq ->
  Kchain_proper (KFK kbr fq)
(* Proof omitted *)
```

10.9.4.3 Ignoring the remainder

Let $n = bq + r$, with $0 < r < b$. The following construction computes x^r and x^b, then x^{bq}, and finally sends x^{bq+r} to the continuation, throwing away x^b.

```
Definition KFF (kbr : Kchain) (fq : Fchain) : Fchain :=
  K2F (KFK kbr fq).
```

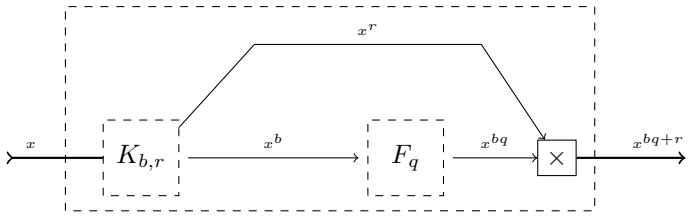


Figure 10.13: The KFF combinator

```
Lemma KFF_correct :
forall (b q r : positive) (kbr : Kchain) (fq : Fchain),
Kchain_correct  b r kbr ->
Fchain_correct q fq ->
Kchain_proper kbr ->
Fchain_proper fq -> Fchain_correct (b * q + r) (KFF kbr fq).
(* Proof omitted *)


Instance KFF_proper :
forall (kbr : Kchain) (fq : Fchain),
Kchain_proper kbr -> Fchain_proper fq -> Fchain_proper (KFF kbr fq).
(* Proof omitted *)
```

10.9.4.4 Conversion of an f-chain into a k-chain

The following conversion is useful when a chain generation algorithm needs to build a k-chain for exponents p and 1:

```
Definition FK (f : Fchain) : Kchain :=
  fun (A : Type) (k : Kkont A) (a : A) =>
    f A (fun y => k y a) a.

Lemma FK_correct : forall (p: positive) (Fp : Fchain),
                     Fchain_correct  p Fp ->
                     Fchain_proper Fp ->
                     Kchain_correct p 1 (FK Fp) .
(* Proof omitted *)

Instance  FK_proper  (Fp : Fchain) (_ : Fchain_proper Fp):
  Kchain_proper (FK Fp).
(* Proof omitted *)
```

10.9.4.5 Computing x^p *and* x^{pq}

```
Definition FFK (fp fq : Fchain) : Kchain :=
  fun A k a =>  fp A (fun xb  => fq A (fun y => k y xb) xb) a.


Lemma FFK_correct  (p q  : positive) (fp fq : Fchain):
    Fchain_correct p fp  ->
```

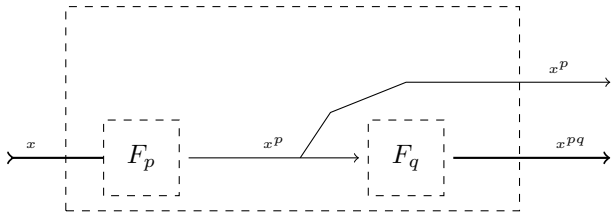


Figure 10.14: The FFK combinator

```
    Fchain_correct q fq ->
    Fchain_proper fp ->
    Fchain_proper fq -> Kchain_correct  (p * q ) p (FFK fp fq).
(* Proof omitted *)

Instance FFK_proper  (fp fq : Fchain)
    (_ :   Fchain_proper fp)
    (_ :  Fchain_proper fq) :  Kchain_proper (FFK fp fq) .
(* Proof omitted *)
```

10.9.4.6 A correct-by-construction chain

A simple example will show us how to build correct chains for any positive exponent, using the operators above.

```
Hint Resolve KFF_correct KFF_proper KFK_correct KFK_proper.

Definition F87 :=
 let k7_3 :=  KFK k3_1 (Fexp2 1) in
 let k10_7 := KFK k7_3 F1 in
 KFF k10_7 (Fexp2 3).

Lemma OK87 : Fchain_correct 87 F87.
Proof.
 unfold F87; change 87 with (10 * (2 ^ 3) + 7)%positive.
 apply KFF_correct;auto.
 change 10 with (7 * 1 + 3); apply KFK_correct;auto.
 change 7 with (3 * 2 ^ 1 + 1)%positive;  apply KFK_correct;auto.
Qed.
```

Note that this method of construction still requires some interaction from the user. In the next section, we build a *function* that maps any positive number n into a correct and proper chain for n. Thus correct chain generation will be fully automated.

10.9.5 Automatic chain generation by Euclidean division

The goal of this section is to write a function `make_chain (p:positive): chain` that builds a correct chain for p, using the Euclidean method above. In other words, we want to get correct chains by computation. The correctness of the result of this computation should be asserted by a theorem:

```
Theorem make_chain_correct :
   forall p, chain_correct p (make_chain p).
```

In the previous section, we considered two different kinds of objects: f-chains, associated with a single exponent, and k-chains, associated with two exponents. We would expect that the function `make_chain` we want to build and certify is structured as a pair of mutually recursive functions. In Coq , various ways of building such functions are available:

- Structural [mutual] recursion with `Fixpoint`
- Using `Program Fixpoint`
- Using `Function`.

Since our construction is based on Euclidean division, we could not define our chain generator by structural recursion. For simplicity's sake, we chose to avoid dependent elimination and used `Function` with a decreasing measure.

For this purpose, we define a single data-type for associated with the generation of F- and K-chains.

We had two slight technical problems to consider:

- The generation of a k-chain for n and p is meaningful only if $p < n$. Thus, in order to avoid a clumsy dependent pattern-matching, we chose to represent a pair (n, p) where $0 < p < n$ by a pair of positive numbers (p, d) where $d = n - p$
- In order to avoid to deal explicitly with mutual recursion, we defined a type called `signature` for representing both forms of function calls. Thus, it is easy to define a decreasing measure on type `signature` for proving termination. Likewise, correctness and properness statements are also indexed by this type.

```
Inductive signature : Type :=
|  (** Fchain for the exponent n *)
     gen_F (n:positive)
| (** Kchain for the exponents p+d  and p *)
     gen_K (p d: positive).
```

The following dependently-typed functions will help us to specify formally any correct chain generator.

```
(**
  exponent associated with a signature:
*)
Definition signature_exponent (s:signature) : positive :=
 match s with
| gen_F n => n
| gen_K p d  =>  p + d
end.
```

```
(**
Type of the associated continuation
*)

Definition kont_type (s: signature)(A:Type) : Type :=
match s with
| gen_F _  => Fkont A
| gen_K _ _   => Kkont A
end.

Definition chain_type (s: signature) : Type :=
 match s with
| gen_F _   => Fchain
|  gen_K _ _  => Kchain
end.
```

```
Definition correctness_statement (s: signature) :
chain_type s -> Prop :=
match s  with
  | gen_F p => fun ch => Fchain_correct p ch
  | gen_K p d   => fun ch => Kchain_correct (p + d) p ch
end.

Definition proper_statement (s: signature) :
chain_type s -> Prop :=
match s  with
  | gen_F p => fun ch => Fchain_proper ch
  | gen_K p d   => fun ch => Kchain_proper ch
end.

(**  Full correctness *)

Definition  OK (s: signature)
  := fun c: chain_type s =>
       correctness_statement s c /\
       proper_statement s c.
```

10.9.6 Generation of chains using Euclidean Division

Assume we want to build automatically a correct f-chain for some positive integer n. If n equals to 1, 3, or 2^p for some positive integer p, this task is immediate, thanks to the constants `F1`, `F3` and `Fexp2`. Otherwise, like in [BCHM95], we decompose n into $bq + r$, where $1 < b < n$, and compose the recursively built chains for q and r on one side, and q on the other side.

The efficiency of this method depends on the choice of b. In [BCHM95], the function that maps n into b is called a *strategy*.

From additions.Dichotomy.

```
Class Strategy (gamma : positive -> positive):=
{
```

```
gamma_lt :forall p:positive, 3 < p -> gamma  p < p;
gamma_gt : forall p:positive, 3 < p -> 1 < gamma  p
}.
```

10.9.7 The dichotomic strategy

In this chapter, we concentrate on the so-called *dichotomic strategy*, defined as follows:

$$n \mapsto n \div 2^k \textbf{ where } k = \lfloor (\log_2 n)/2 \rfloor$$

Intuitively, it corresponds to splitting the binary representation of a positive integer into two halves. For instance, consider $n = 87$ its binary representation is 1010111. The number $\lfloor (\log_2 n)/2 \rfloor$ is equal to 3. Dividing n by 2^3 gives the decomposition $n = 10 \times 2^3 + 7$. Thus, a chain for $n = 87$ can be built from a chain computing both x^7 and x^{10}, and a chain that raises its argument to its $8 - th$ power.

This strategy is defined in Module additions.Dichotomy.

```
Function dicho_aux (p:positive) {struct p} : positive :=
 match p with
   | 1%positive   =>  xH
   | 2%positive |   3%positive  => 2
   | xO (xO q) | xO (xI q) | xI (xO q) | xI (xI q) =>
                                        xO (dicho_aux q)
 end.

Definition dicho  (p:positive) : positive :=
  N2pos (N.div (Npos p) (Npos (dicho_aux p))).

Instance Dicho_strat : Strategy dicho.
```

10.9.8 Other strategies

For comparison's sake, we define two other strategies, much simpler but statically less efficient than the dichotomic strategy.

From Module additions.BinaryStrat.

```
Definition half (p:positive) :=
  match p with xH => xH
          |    xI q | xO q =>  q
  end.

Definition two (p:positive) := 2%positive.

Instance Binary_strat : Strategy half.
Proof.
  split; destruct p; unfold half; try lia.
Qed.

Instance Two_strat : Strategy two.
```

```
Proof.
  split;unfold two; lia.
Qed.
```

Page. 253, we compare the three strategies with respect to the length of the built chains.

10.9.9 Main chain generation function

We are now able to define a function that generates a correct chain for any signature. We use the `Recdef` module of Standard Library, with an appropriate *measure*.

```
Definition signature_measure (s : signature) : nat :=
match s with
  | gen_F n => 2 * Pos.to_nat n
  | gen_K p d => 2 * Pos.to_nat (p + d) +1
end.
```

The following function definition generates 9 subgoals, for proving that the measure on signatures is strictly decreasing along the recursive calls. They are solved with the help of Standard Library's lemmas on arithmetic of `positive` numbers and Euclidean division.

```
Function chain_gen  (s:signature) {measure signature_measure}
:  chain_type s :=
  match s  return chain_type s with
    | gen_F i =>
      if pos_eq_dec i 1 then F1 else
        if pos_eq_dec i 3
        then F3
        else
          match exact_log2 i with
              Some p => Fexp2 p
            | _ =>
              match N.pos_div_eucl i (Npos (dicho i))
              with
                | (q, 0%N) =>
                  Fcompose  (chain_gen (gen_F (dicho i)))
                            (chain_gen (gen_F (N2Pos q)))
                | (q,r)  => KFF (chain_gen
                                   (gen_K (N2Pos r)
                                          (dicho i - N2Pos r)))
                                (chain_gen (gen_F (N2Pos q)))

              end end
```

```
    | gen_K p d =>
      if pos_eq_dec p 1 then FK (chain_gen (gen_F (1 + d)))
      else
        match N.pos_div_eucl (p + d)  (Npos p) with
          | (q, 0%N) => FFK   (chain_gen (gen_F p))
```

```
                              (chain_gen (gen_F (N2Pos q)))
          | (q,r)  => KFK (chain_gen (gen_K (N2Pos r)
                                            (p - N2Pos r)))
                          (chain_gen (gen_F (N2Pos q)))
        end
  end.
(* A lot of arithmetic proofs omitted *)
Defined.

Definition make_chain (n:positive) : chain :=
 F2C (chain_gen (gen_F n)).
```

Thanks to the `Recdef` package, we are now able to get automatically built chains using the dichotomic strategy.

```
Compute make_chain 87.
```

```
 =  fun (A : Type) (x : A) =>
       x0 <--- x times x;
       x1 <--- x0 times x;
       x2 <--- x1 times x1;
       x3 <--- x2 times x;
       x4 <--- x3 times x1;
       x5 <--- x4 times x4;
       x6 <--- x5 times x5;
       x7 <--- x6 times x6;
       x8 <--- x7 times x3;
       Return x8
     : chain
```

10.9.9.1 A few tests

The following tests show various examples of chains for the same exponent, using different strategies. The dichotomic strategy seems clearly to be the winner (at least on this sample).

```
Compute chain_length (make_chain two 56789).
```

```
= 25%nat : nat
```

```
Compute chain_length (make_chain half 56789).
```

```
 = 25%nat : nat
```

```
Compute chain_length (make_chain dicho 56789).
```

```
= 21%nat : nat
```

```
Compute chain_length (make_chain two 3456789).
```

```
= 33%nat : nat
```

```
Compute chain_length (make_chain half 3456789).
```

```
(= 33%nat : nat
```

```
Compute chain_length (make_chain dicho 3456789).
```

```
= 29%nat : nat
```

10.9.9.2 Correctness of the Euclidean chain generator

`Recdef`'s `functional induction` tactic allows us to prove that every value returned by (`chain_gen` s) is correct w.r.t. s and proper. The proof obligations are solved thanks to the previous lemmas on the composition operators on chains: `Fcompose`, `KFK`, etc. Unfortunately, a lot of interaction is still needed for proving properties of Euclidean division and binary logarithm.

```
Lemma chain_gen_OK : forall s:signature, OK  s (chain_gen  s).
intro s; functional induction chain_gen s.
Proof.
(* A lot of arithmetic proofs omitted *)

Theorem make_chain_correct :
    forall p, chain_correct p (make_chain p).
Proof.
 intro p; destruct (chain_gen_OK (gen_F p)).
 unfold make_chain; apply F2C_correct; apply H.
Qed.
```

10.9.9.3 A last example

Let us compute $67777^{6145319}$ with 32 bits integers:

```

Ltac compute_chain ch :=
   let X := fresh "x" in
   let Y := fresh "y" in
   let X := constr:ch in
   let Y := (eval vm_compute in X) in
   exact Y.

Let big_chain := ltac:(compute_chain  (make_chain 6145319)).

Print big_chain.
```

```
big_chain =
fun (A : Type) (x : A) =>
x0 <--- x times x; x1 <--- x0 times x0;
x2 <--- x1 times x1; x3 <--- x2 times x1;
x4 <--- x3 times x3; x5 <--- x4 times x;
x6 <--- x5 times x5; x7 <--- x6 times x6;
x8 <--- x7 times x1; x9 <--- x8 times x5;
x10 <--- x9 times x8; x11 <--- x10 times x9;
x12 <--- x11 times x11; x13 <--- x12 times x11;
x14 <--- x13 times x10; x15 <--- x14 times x14;
x16 <--- x15 times x11; x17 <--- x16 times x16;
x18 <--- x17 times x17; x19 <--- x18 times x18;
x20 <--- x19 times x19; x21 <--- x20 times x20;
x22 <--- x21 times x21; x23 <--- x22 times x22;
x24 <--- x23 times x23; x25 <--- x24 times x24;
x26 <--- x25 times x25; x27 <--- x26 times x26;
x28 <--- x27 times x14;  Return x28
     : forall A : Type, A -> computation
```

```
Time   Compute  Int31.phi
     (chain_apply big_chain (snd (positive_to_int31  67777))).
```

```
= 2014111041%Z
     : Z
Finished transaction in 0.005 secs (0.005u,0.s) (successful)}
```

```
Compute chain_length big_chain.
```

```
= 29%nat
     : nat
```

10.9.10 Fibonacci, *le retour*

It is now possible to use Euclidean addition chains for computing Fibonacci numbers (see Sections 10.2.3.2 on page 198 and 10.4.6 on page 212).

The following function is parameterized by any strategy γ.

```
Definition fib_eucl gamma `{Hgamma: Strategy gamma} n :=
  let c := make_chain gamma  n
  in let r := chain_apply c (M:=Mul2) (1,0) in
       fst r + snd r.

Compute fib_eucl dicho 153.
```

```
    = 68330027629092351019822533679447
     : N
Finished transaction in 0.002 secs (0.002u,0.s) (successful)
```

```
Compute fib_eucl two 153.
```

```
    = 68330027629092351019822533679447
     : N
Finished transaction in 0.003 secs (0.003u,0.s) (successful)
```

```
Compute fib_eucl half 153.
```

```
    = 68330027629092351019822533679447
     : N
Finished transaction in 0.003 secs (0.003u,0.s) (successful)
```

10.10 Projects

Project 10.3 (Optimality and relative efficiency)

1. Prove that the chain generated by `Fexp2` is optimal.
2. Prove that the length of any optimal chain for n is greater than or equal to $\lfloor \log_2 n \rfloor$.
3. Prove that, for any positive n, the length of any Euclidean chain generated by the dichotomic strategy is always less than or equal to the length of `binary_chain` n, and for an infinite number of positive integers n, the first chain is strictly shorter than the latter.
4. Prove that our implementation of the dichotomic strategy describes the same function as in the literature (for instance [BCHM95].) This is important if we want to follow the complexity analyses in this and similar articles.
5. Study how to *compile* a chain into imperative code, using a register allocation strategy (it may be useful to define *chain width*).

   Remark: The first two questions of the list above should involve a universal quantification on type `chain`. It may be necessary (but we're not sure) to consider some restriction on parametric chains.

10.10.1 A data structure for Euclidean chains

Figures 10.5 on page 234 to 10.14 on page 248 suggest that any computation following an Euclidean chain can be executed on a kind of abstract machine with a "register" and a stack, and only four operations:

- multiply the contents of the register by the top of the stack (and pop that stack),
- raising the contents of the register to its square,
- push the contents of the register into the stack,
- swapping the two elements at the top of the stack.

In Coq, we define the instructions as the four constructors of an inductive type.

From Module additions.AM

```
(** basic instructions *)

Inductive instr : Set :=
  | MUL : instr
  | SQR : instr
  | PUSH : instr
  | SWAP : instr.

Definition code := list instr.

(* semantics *)
(*************)

Section Semantics.

 Variable A : Type.
 Variable mul : A -> A -> A.
 Variable one : A.

 Definition stack := list A.
 Definition config := (A * list A)%type.

 Fixpoint exec (c : code) (x:A) (s: stack) : option config :=
   match c, s with
     nil, _ => Some (x,s)
   | MUL::c, y::s => exec c (mul x y) s
   | SQR::c, s => exec c (mul x x) s
   | PUSH::c, s => exec c x (x::s)
   | SWAP::c, y::z::s => exec c x (z::y::s)
   | _,_ => None
   end.

(* ... *)
End Semantics.
```

For instance the chain of Fig. 10.4 on page 217 can be represented with the following code:

```
  PUSH :: PUSH :: SQR  :: MUL :: PUSH :: SWAP :: SQR :: MUL :: PUSH
           :: SWAP :: MUL :: SQR :: SQR :: SQR :: MUL :: nil
```

In the library additions.AM, we define a chain generator for this data structure. Please note that many proof scripts are copied verbatim from `Euclidean_Chains` into `AM`. Removing such redundancies is left as a project.

Project 10.4 (Some improvements) 1. Improve automated proofs on types `positive` and `N`.

2. Compare `Program Fixpoint` and `Function` for writing `make_chain`. Consider measure *vs* well-founded relations, mutual recursion, possibility of using sigma-types, etc.

3. Chains are always associated with strictly positive exponents. Thus, many lemmas about chain correctness can be proved using semi-groups instead of monoids. Define type classes for semi-groups and use them whenever possible.

Part III

Appendices

Bibliography

[Abr96] Jean-Raymond Abrial. *The B-book: Assigning Programs to Meanings*. Cambridge University Press, New York, NY, USA, 1996.

[Abr10] Jean-Raymond Abrial. *Modeling in Event-B: System and Software Engineering*. Cambridge University Press, New York, NY, USA, 1st edition, 2010.

[Bau] Andrej Bauer. The hydra game. `http://math.andrej.com/2008/02/02/the-hydra-game`.

[Bau08] Andrej Bauer. The hydra game source code. `https://github.com/andrejbauer/hydra`, 2008.

[BB87] Jean Berstel and Srećko Brlek. On the length of word chains. *Information Processing Letters*, 26(1):23–28, 1987. `http://www-igm.univ-mlv.fr/~berstel/Articles/1987WordChains.pdf`.

[BC04] Yves Bertot and Pierre Castéran. *Interactive Theorem Proving and Program Development: Coq'Art: The Calculus of Inductive Constructions*. Springer, Berlin, Heidelberg, 2004. `https://www.labri.fr/perso/casteran/CoqArt/`.

[BCHM95] Srećko Brlek, Pierre Castéran, Laurent Habsieger, and Richard Mallette. On-line evaluation of powers using Euclid's algorithm. *RAIRO - Theoretical Informatics and Applications - Informatique Théorique et Applications*, 29(5):431–450, 1995. `http://www.numdam.org/item/ITA_1995__29_5_431_0.pdf`.

[BCS91] Srećko Brlek, Pierre Castéran, and Robert Strandh. On addition schemes. In *International Joint Conference on Theory and Practice of Software Development*, pages 379–393, Berlin, Heidelberg, 1991. Springer. `https://link.springer.com/content/pdf/10.1007%2F3540539816_77.pdf`.

[Bra39] Alfred Brauer. On addition chains. *Bulletin of the American Mathematical Society*, 45(10):736–739, 10 1939. `https://www.ams.org/journals/bull/1939-45-10/S0002-9904-1939-07068-7/S0002-9904-1939-07068-7.pdf`.

[Bro10] Daniel Brown. Parametricity. `https://web.archive.org/web/20190628092255/http://www.ccs.neu.edu/home/matthias/369-s10/Transcript/parametricity.pdf`, 2010. Transcript of a lecture by Matthias Felleisen.

[Bur75] William H. Burge. *Recursive programming techniques*. Addison-Wesley, 1975.

[Can55] Georg Cantor. *Contributions to the Founding of the Theory of Transfinite Numbers*. Courier Corporation, 1955.

[Cas04] Pierre Castéran. Additions. User Contributions to the Coq Proof Assistant, 2004. `https://github.com/coq-contribs/additions`.

[Cas07] Pierre Castéran. Utilisation en Coq de l'opérateur de description. In *Actes des Journées Francophones des Langages Applicatifs*, pages 30–44, 2007. `http://jfla.inria.fr/2007/actes/PDF/03_casteran.pdf`.

[CC06] Pierre Castéran and Évelyne Contejean. On ordinal notations. User Contributions to the Coq Proof Assistant, 2006. `https://github.com/coq-contribs/cantor`.

[CDM13] Cyril Cohen, Maxime Dénès, and Anders Mörtberg. Refinements for free! In *Certified Programs and Proofs*, pages 147–162, Cham, 2013. Springer. `https://hal.inria.fr/hal-01113453`.

[Chl08] Adam Chlipala. Parametric higher-order abstract syntax for mechanized semantics. In *International Conference on Functional Programming*, pages 143–156, New York, NY, USA, 2008. ACM. `http://adam.chlipala.net/papers/PhoasICFP08/`.

[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 Jouannaud on the Occasion of His 60th Birthday*. Springer, Berlin, Heidelberg, 2007. `https://link.springer.com/book/10.1007%2F978-3-540-73147-4`.

[CPU+10] Évelyne Contejean, Andrei Paskevich, Xavier Urbain, Pierre Courtieu, Olivier Pons, and Julien Forest. A3PAT, an approach for certified automated termination proofs. In *Workshop on Partial Evaluation and Program Manipulation*, pages 63–72, New York, NY, USA, 2010. Association for Computing Machinery. `https://hal.inria.fr/inria-00535655`.

[CS] Pierre Castéran and Matthieu Sozeau. A gentle Introduction to Type Classes and Relations in Coq. `https://www.labri.fr/perso/casteran/CoqArt/TypeClassesTut/typeclassestut.pdf`.

[Der82] Nachum Dershowitz. Orderings for term-rewriting systems. *Theoretical Computer Science*, 17(3):279–301, 1982. `https://www.sciencedirect.com/science/article/pii/0304397582900263/pdf`.

[DM07] Nachum Dershowitz and Georg Moser. The hydra battle revisited. In *Rewriting, Computation and Proof: Essays Dedicated to Jean-Pierre Jouannaud on the Occasion of His 60th Birthday*, pages 1–27. Springer, Berlin, Heidelberg, 2007. https://www.cs.tau.ac.il/~nachum/papers/LNCS/Hydra.pdf.

[GAA+13] Georges Gonthier, Andrea Asperti, Jeremy Avigad, Yves Bertot, Cyril Cohen, François Garillot, Stéphane Le Roux, Assia Mahboubi, Russell O'Connor, Sidi Ould Biha, Ioana Pasca, Laurence Rideau, Alexey Solovyev, Enrico Tassi, and Laurent Théry. A machine-checked proof of the odd order theorem. In *Interactive Theorem Proving*, pages 163–179, Berlin, Heidelberg, 2013. Springer. https://hal.inria.fr/hal-00816699.

[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. *Annals of Pure and Applied Logic*, 53(3):199–260, 1991. https://www.sciencedirect.com/science/article/pii/016800729190022E/pdf.

[GM93] M. J. C. Gordon and T. F. Melham, editors. *Introduction to HOL: A theorem proving environment for higher order logic*. Cambridge University Press, 1993. http://www.cs.ox.ac.uk/tom.melham/pub/Gordon-1993-ITH.html.

[Gon08] Georges Gonthier. Formal proof—the four-color theorem. *Notices of the American Mathematical Society*, 55(11):1382–1393, 2008. http://www.ams.org/notices/200811/tx081101382p.pdf.

[Goo44] R. L. Goodstein. On the restricted ordinal theorem. *Journal of Symbolic Logic*, 9(2):33–41, 1944. https://www.jstor.org/stable/2268019.

[Gri13] José Grimm. Implementation of three types of ordinals in Coq. Research Report RR-8407, INRIA, 2013. https://hal.inria.fr/hal-00911710.

[HAB+17] Thomas Hales, Mark Adams, Gertrud Bauer, Tat Dat Dang, John Harrison, Le Truong Hoang, Cezary Kaliszyk, Victor Magron, Sean McLaughlin, Tat Thang Nguyen, Quang Truong Nguyen, Tobias Nipkow, Steven Obua, Joseph Pleso, Jason Rute, Alexey Solovyev, Thi Hoai An Ta, Nam Trung Tran, Thi Diep Trieu, Josef Urban, Ky Vu, and Roland Zumkeller. A formal proof of the Kepler conjecture. *Forum of Mathematics, Pi*, 5, 2017. https://arxiv.org/abs/1501.02155.

[Hue97] Gérard Huet. The zipper. *Journal of Functional Programming*, 7(5):549–554, September 1997. https://www.cambridge.org/core/journals/journal-of-functional-programming/article/zipper/0C058890B8A9B588F26E6D68CF0CE204.

[KP82] Laurie Kirby and Jeff Paris. Accessible independence results for Peano arithmetic. *Bulletin of the London Mathematical Society*, 14(4):285–293, 1982. https://faculty.baruch.cuny.edu/lkirby/accessible_independence_results.pdf.

[KS81] Jussi Ketonen and Robert Solovay. Rapidly growing Ramsey functions. *Annals of Mathematics*, 113(2):267–314, 1981. http://www.jstor.org/stable/2006985.

[MT18] Assia Mahboubi and Enrico Tassi. Mathematical Components. https://doi.org/10.5281/zenodo.3999478, 2018. With contributions by Yves Bertot and Georges Gonthier.

[MV05] Panagiotis Manolios and Daron Vroon. Ordinal arithmetic: Algorithms and mechanization. *Journal of Automated Reasoning*, 34(4):387–423, May 2005. http://www.ccs.neu.edu/home/pete/pub/ordinal-arithmetic-algs-mech.pdf.

[NPW02] Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. *Isabelle/HOL: A Proof Assistant for Higher-Order Logic*. Springer, Berlin, Heidelberg, 2002. https://link.springer.com/book/10.1007%2F3-540-45949-9.

[O'C05] Russell O'Connor. Essential incompleteness of arithmetic verified by Coq. In *International Conference on Theorem Proving in Higher Order Logics*, pages 245–260, Berlin, Heidelberg, 2005. Springer. https://arxiv.org/abs/cs/0505034.

[P$^+$] Benjamin Pierce et al. Software foundations. https://softwarefoundations.cis.upenn.edu.

[PC] Clément Pit-Claudel. Alectryon. https://github.com/cpitclaudel/alectryon.

[PC20] Clément Pit-Claudel. Untangling mechanized proofs. In *International Conference on Software Language Engineering*, pages 155–174, New York, NY, USA, 2020. Association for Computing Machinery. https://dl.acm.org/doi/pdf/10.1145/3426425.3426940.

[Pla13] PlanetMath. Ackermann function is not primitive recursive. https://planetmath.org/ackermannfunctionisnotprimitiverecursive, 2013.

[Prö13] Hans Jürgen Prömel. Rapidly growing ramsey functions. In *Ramsey Theory for Discrete Structures*, pages 97–103. Springer, Cham, 2013. https://link.springer.com/chapter/10.1007/978-3-319-01315-2_8.

[Rey83] John C. Reynolds. Types, abstraction and parametric polymorphism. In R. E. A. Mason, editor, *Information Processing 83*, pages 513–523, Amsterdam, 1983. Elsevier.

[Rey93] John C. Reynolds. The discoveries of continuations. *LISP and Symbolic Computation*, 6:233–247, 1993. https://link.springer.com/content/pdf/10.1007/BF01019459.pdf.

[Sch77] Kurt Schütte. *Proof Theory*. Springer, 1977. https://link.springer.com/book/10.1007%2F978-3-642-66473-1.

[Sla07] Will Sladek. The Termite and the Tower: Goodstein sequences and provability in PA. https://www.uio.no/studier/emner/matnat/ifi/INF5170/v08/undervisningsmateriale/sladekgoodstein.pdf, 2007.

[SM19] Matthieu Sozeau and Cyprien Mangin. Equations reloaded: High-level dependently-typed functional programming and proving in Coq. *Proceedings of the ACM on Programming Languages*, 3(ICFP), July 2019. https://hal.inria.fr/hal-01671777.

[SO08] Matthieu Sozeau and Nicolas Oury. First-class type classes. In *International Conference on Theorem Proving in Higher Order Logics*, pages 278–293, Berlin, Heidelberg, 2008. Springer. https://sozeau.gitlabpages.inria.fr/www/research/publications/First-Class_Type_Classes.pdf.

[Str00] Christopher Strachey. Fundamental concepts in programming languages. *Higher-Order and Symbolic Computation*, 13(1-2):11–49, April 2000. https://www.cs.cmu.edu/~crary/819-f09/Strachey67.pdf.

[SvdW11] Bas Spitters and Eelis van der Weegen. Type classes for mathematics in type theory. *Mathematical Structures in Computer Science*, 21(4):795–825, 2011. https://arxiv.org/abs/1102.1323.

[Tel00] Gerard Tel. *Introduction to Distributed Algorithms*. Cambridge University Press, Cambridge, United Kingdom, second edition, 2000.

[The] The Coq Development Team. The Coq Proof Assistant. https://coq.inria.fr.

[Wad89] Philip Wadler. Theorems for free! In *International Conference on Functional Programming Languages and Computer Architecture*, pages 347–359, New York, NY, USA, 1989. ACM. https://homepages.inf.ed.ac.uk/wadler/papers/free/free.ps.

[Wai70] S. S. Wainer. A classification of the ordinal recursive functions. *Archiv für mathematische Logik und Grundlagenforschung*, 13(3):136–153, Dec 1970. https://link.springer.com/article/10.1007%2FBF01973619.

[WB87] Stan Wainer and Wilfried Buchholz. Provably computable functions and the fast growing hierarchy. In Stephen G. Simpson, editor, *Contemporary Mathematics*, volume 65, pages 179–198. American Mathematical Society, Providence, RI, USA, 1987. http://nbn-resolving.de/urn/resolver.pl?urn=nbn:de:bvb:19-epub-3843-7.

Chapter 11

Index and tables

In progress This index is currently under reorganization for a few days. We apologize for its incompleteness!

Coq, plug-ins and standard library

Mathematical notions and algorithmics

Library hydras: Ordinals and hydra battles

Library hydras.Ackermann: Primitive recursive functions

Library additions: Addition chains

11.1 Main notations

Table 11.1: Ordinals and ordinal notations

Name	Gallina	Math	Description	Page
`lt : T1->T1->Prop`	lt alpha beta	$\alpha < \beta$	strict order on type `T1` [1]	76
`LT: T1->T1->Prop`	alpha o< beta	$\alpha < \beta$	strict order on type `T1` [2]	78
`Lt : E0->E0->Prop`	alpha o< beta	$\alpha < \beta$	strict order on type `E0` [3]	78
`nf: T1->Prop`	`nf alpha`		alpha is in Cantor normal form	77
`on_lt`	`alpha o< beta`	$\alpha < \beta$	ordinal inequality [4]	51
`on_le`	`alpha o<= beta`	$\alpha \leq \beta$	ordinal inequality	51
`plus`	`alpha + beta`	$\alpha + \beta$	ordinal addition	81, …
`oplus`	`alpha o+ beta`	$\alpha \oplus \beta$	Hessenberg sum	88
F	`F` n	n	The n-th finite ordinal	73, 142
FS	`FS` n	$n+1$	The $n+1$-th finite ordinal [5]	73
omega	`omega`	ω	the first infinite ordinal	143, 79, 74, …
phi0	`phi0 alpha`	$\phi_0(\alpha)$, ω^α	exponential of base ω	74

[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.
[4] 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.
[5] 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.

Table 11.2: hydra battles

Name	Gallina	Math	Description	Page
`round`	`h -1-> h'`		one round of a battle	31
`rounds`	`h -+-> h'`		one or more rounds of a battle	31
`round_star`	`h -*-> h'`		any number of rounds of a battle	31

Table 11.3: Addition chains

Name	Gallina	Math	Description	Page
`Mult`	z `<---` x `times` y		monadic notation	216