Le Teaser

Pierre Castéran
LaBRI, Univ. Bordeaux, CNRS UMR 5800, Bordeaux-INP

October 23, 2019

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

Contents

Chapter 1

Introduction

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

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

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

This is obviously the case of the four-color theorem [24] and the Kepler conjecture [23].

Structure of this document

- We present several contributions, whose topic is easy to understand. Each contribution is chosen according to its potential to illustrate interesting proof patterns, or how to use some libraries of the Coq system
- Whenever several implementations are possible, we will discuss the pros and cons of every possible choice
- 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 143, the notion

of *limit ordinal* is made more "concrete" thanks to a function `canonseq` that 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 $\omega^{\omega^{\omega}}$!

- We found it interesting to present several implementions of a given concept. After some discussions of the pros and cons of each solution, we will choose to 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.

Warning: This document is *not* an introductory text for Coq, and there are many aspects of this proof assistant that are not covered. The reader should already have some basic experience with the Coq system. The Reference Manual and several tutorials are available on Coq page [21]. First chapters of textbooks like *Interactive Theorem Proving and Program Development* [5], *Software Foundations* [31] or *Certified Programming with Dependent Types* [17] will give you the right background.

Contributions are welcome

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

1.0.0.1 Acknowledgements

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

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

I owe my interest in discrete maths and their relation to formal proofs and functional programmings to Srecko Brlek. Equally, there is W. H. Burge's book "*Recursive Programming Techniques*" [10] which was a great source of inspiration.

1.0.0.2 Typographical conventions

Quotations from our Coq source are displayed as follows:

```
Require Import Arith.

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

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

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

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

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

```
   reflexivity.
Qed.
```

1.0.0.3 Alternative or bad definitions

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

```
Module Bad.

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

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

End Bad.
```

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

```
Module Alt.

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

End Alt.
```

```
Lemma alt_double_ok n : Nat.double n = Alt.double n.
Proof.
  unfold Alt.double, Nat.double.
  omega.
Qed.
```

1.0.0.4 Links to the Coq source

Active links towards our Coq modules may be incorrect if you got this `pdf` document otherwise than by compiling the distribution available in `http://www.labri.fr/casteran/CoqArt/le_teaser/Teaser.tar.gz`.

Part I

Some Certified Algorithms

Chapter 2

Smart Computation of x^n

2.1 Introduction

Nothing looks simpler than writing some function for computing x^n. On the contrary, this simple programming exercice allows us to address such advanced programming techniques as:

- monadic programming, and continuation passing style
- type classes, and generalized rewriting
- proof engineering, in particular proof re-using
- proof by reflection
- polymorphism and parametricity
- composition of correct programs, etc.

2.2 Some basic implementations

Let us start with a very naïve 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 *Powers.FirstSteps.v*

```
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
```

This function is linear with respect to the number of multiplications needed to compute x^n. 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`.

2.2.1 A semi-naïve algorithm

In versions up to `V8.9.1`, the exponentiation function on type `Z` was defined as follows, (in modules `Coq.PArith.BinPosDef.Pos` and `Coq.ZArith.BinIntDef.Z`.

```
(** ** Iteration of a function over a positive number *)

Definition iter {A} (f:A -> A) : A -> positive -> A :=
  fix iter_fix x n := match n with
    | xH => f x
    | xO n' => iter_fix (iter_fix x n') n'
    | xI n' => f (iter_fix (iter_fix x n') n')
  end.

Definition pow (x:positive) := iter (mul x) 1.
```

```
Definition pow_pos (z:Z) := Pos.iter (mul z) 1.

Definition pow x y :=
  match y with
    | pos p => pow_pos x p
```

```
    | 0 => 1
    | neg _ => 0
  end.

Infix "^" := pow : Z_scope.
```

At first sight, the function `Pos.pow` seems to be logarithmic because of the recursive structure of the help function `iter_fix`. Unfortunately, it is obvious that a call to `iter f x n` will apply n times the function f. Thus, these exponentiation functions with binary exponents are in fact linear!

```
Time Compute (1 ^ 56666667)%N.
```

```
Finished transaction in 3.604 secs (3.587u,0.007s)
```

2.2.2 A truly logarithmic exponentiation function

Using the following equations, we can easily define a polymorphic exponentiation whose application requires only a logarithmic number of multiplications.

$$x^1 = x \tag{2.1}$$
$$x^{2p} = (x^2)^p \tag{2.2}$$
$$x^{2p+1} = (x^2)^p \times x \tag{2.3}$$
$$x^1 \times a = x \times a \tag{2.4}$$
$$x^{2p} \times a = (x^2)^p \times a \tag{2.5}$$
$$x^{2p+1} \times a = (x^2)^p \times (a \times x) \tag{2.6}$$

In equalities 2.4 to 2.6, the variable a plays the rôle of an *accumulator* whose initial value (set by 2.3) 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 2.2 on page 15, we can implement this algorithm in Gallina. The following definitions are still within the scope of the section open in 2.2 on page 15.

From FirstSteps.v

```
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} _ _ _ _.
```

2.2.2.0.1 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. 2.3.1 on page 22).

2.2.3 Examples of computation

It is now possible to test our functions with various interpretations of $\times$ and 1:

```
Compute power 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
```

2.2.3.1 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 [21].

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. We use the infix notation `**` for distinguishing matrix multiplication from multiplication on type `A`. Note that the use of operational type classes in Sect 2.3.1 will make this distinction useless.

```
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').

Infix "'**'  := M2_mult (at level 40, left associativity).
```

The neutral element for `M2_mult` is the identity matrix.

```
Definition Id2 : t := mat 1 0 0 1.
```

```
End M2_Definitions.
End M2.
```

Matrix exponentiation is a well-known method for computing Fibonacci numbers:

```
Import M2.

Arguments M2_mult {A} plus mult  _  _.
Arguments mat {A} _ _ _ _.
Arguments Id2 {A}  _ _.

Definition fibonacci (n:N) :=
 c00 N  (N_bpow  (M2_mult Nplus Nmult)
                 (Id2  0%N 1%N)
                 (mat  1 1 1 0)%N
                 n).

Compute fibonacci 20.
```

```
= 10946%N
     : N
```

2.2.3.2 Remark

Our function `N_bpow` is really logarithmic. Let us make a comparative test with Standard Library's exponentiation function on type `N` (see section 2.2.1 on page 17).

```
Time Compute (N_bpow N.mul 1 1 56666667)%N.
```

```
Finished transaction in 0. secs (0.u,0.s) (successful)
```

2.2.4 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 "naïve" 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 restrictive. 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 *monoïds*. An exponentiation function will be considered as correct if it returns always the same result as `power` *in any monoïd*.

2.3 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:

$$\forall x\, y\, z : A, x \times (y \times z) = (x \times y) \times z) \tag{2.7}$$

$$\forall x : A, x \times \mathbb{1} = \mathbb{1} \times x = x \tag{2.8}$$

In Coq, we define the monoid structure in terms of *operational type classes*[37, 36]. The tutorial on type classes [15] 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.

2.3.1 A common notation for multiplication

Operational type classes (Spitters and van der Weegen [37]) 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.

```
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 to *op*. The practical interest of this definition is that the term `@mult_op A` *op* `x y`, although convertible with *op* `x y`, bears more information than the second one.

We are now ready for defining 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 "*" := binop : 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.

2.3.1.1 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 `binop`. Nevertheless this term is convertible with `Nat.mul 3 4`, as shown by the interaction below.

```
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
```

2.3.1.2 String Concatenation

We can use the notation `"_ * _ "` for other types than numbers. In the following example, the expression `"abc" * "def"` is interpreted as `@binop string ?X "abc" "def"`, then the type class mechanism replaces the unknown ?X with `string_op`.

```
Require Import String.
Instance string_op : Mult_op string := append.
Open Scope string_scope.

Compute "abc" * "def".
```

```
    = "abcdef"  : string
```

2.3.1.3 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 expliciting some arguments of `binop`. For instance, in Sect. 2.3.3.2 on the next page, we consider various monoids on types `nat` and `N`.

2.3.2 The Monoid Type Class

We are now ready for giving a definition of the `Monoid` class, using `*` as an infix operator in scope %M for the monoid multiplication.

The following class definition is parameterized with some type A, a multiplication (called `op` in the definition), and a neutral element $\mathbb{1}$ (called `one` in the definition).

From file ../V8.9/Powers/Monoid_def.v.

```
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
}.
```

2.3.3 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 ../V8.9/Powers/Monoid_instances.v.

2.3.3.1 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.
```

2.3.3.2 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. 2.6.4 on page 47 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 `NPlus` and `NMult` are built for type `N`, and `PMult` for type `positive`.

2.3.3.3 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.
```

2.3.4 Matrices on a semi-ring

In Sect. 2.2.3.1 on page 19, 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 an 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.
```

2.3.5 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 | 0 < 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 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:

```
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`.

```
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:A, x * (y * z) == x * y * z;
    Eone_left : forall x,  E_one * x == x;
    Eone_right : forall x,  x * E_one ==  x
  }.
```

2.3.5.1 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.

Coercion Monoid_EMonoid : Monoid >-> EMonoid.
```

Every instance of `Monoid` can now be considered as an instance of `EMonoid`:

```
Check NMult : EMonoid  N.mul 1%N eq.
```

```
  NMult:EMonoid N.mul 1%N eq
     : EMonoid N.mul 1%N eq
```

2.3.5.2 Example : Arithmetic modulo m

The following instance of `EMonoid` describes the set of integers modulo m, where m is any integer greater or equal than 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.
  Local Open Scope M_Scope.

  Global Instance mod_Equiv : Equivalence mod_equiv.


  Global Instance mult_mod_proper :
  Proper (mod_equiv ==> mod_equiv ==> mod_equiv)  mod_op.
  (* Proof omitted *)

  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.
```

2.3.5.2.1 Example In the following interaction, we show how to instanciate 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
```

2.4 Computing Powers in any EMonoid

The module Pow defines two functions for exponentiation on any `EMonoid` on carrier A. They are essentially the same as in Sect. 2.2 on page 15. 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.

2.4.1 The naïve (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.
```

2.4.1.0.1 Examples of computation From File ../V8.9/Powers/Demo_power.v

The first interaction shows an exponentiation in $\mathbb{Z}$, and the second one in the type of 31 bits machine integers.[1]

[1] `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`.

```
Open Scope M_scope.

Compute 22%Z ^ 20.
```

```
= 70542949868640404420794777 6%Z
```

```
Import Int31.
Coercion phi_inv : Z >-> int31.

Compute (22%int31 ^ 20).
```

```
   = 2131755008%int31
     : int31
```

2.4.2 The Binary Exponentiation Algorithm

Please find below the implementation of binary exponentiation using type classes (to be compared with the version in 2.2.2 on page 17).

```
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 E_op (x * x) a q
    | xI q =>  binary_power_mult E_op (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  E_op (x * x) x q
end.
```

It is easy to extend `Pos_bpow`'s domain to the type of all natural numbers:

```
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.
```

2.4.3 Refinement and Correctness

We have got two functions for computing powers in any monoïd. So, it is interesting to ask oneself whether this duplication is useful, and which would be the respective rôle 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`).

```
Lemma N_bpow_ok :
forall (x:A) (n:N),   x ^b n  == x ^ N.to_nat n.
```

The relationship between `power` and `N_bpow` can be considered as a kind of *refinement* as in the B-method [1]. Note that the two representations of natural numbers and the function `N.to_nat` form a kind of *data refinement* [18, 2].

2.4.4 Proof of correctness of binary exponentiation w.r.t. the function `power`

Section `M_given` of module coq.Exponentiation.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).
```

2.4.4.1 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 A E_op E_one equiv M) ||
```

```
    rewrite (@Eone_right A E_op E_one  equiv M) ||
    rewrite (@Eop_assoc A E_op E_one equiv M).

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_x_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* commutativity of `*`.

```
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 exponentiaton 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 *)
```

2.4.5 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.

```
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:positive)(x:A), Pos_bpow x p == x ^ Pos.to_nat p.
(* Proof omitted *)

Lemma N_bpow_ok :
forall (x:A) (n:N),  x ^b n  == x ^ N.to_nat n.
(* Proof omitted *)
```

```
Lemma N_bpow_ok_R :
forall (x:A) (n:nat), x ^b (N.of_nat n)   ==  x ^  n.
(* Proof omitted *)

Lemma Pos_bpow_ok_R :
forall (x:A) (n:nat), n <> 0 ->
                      Pos_bpow x  (Pos.of_nat n)   ==  x ^  n.
(* Proof omitted *)

End About_power.
```

2.4.5.1 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.
```

2.5 Comparing Exponentiation Algorithms with respect to Efficiency

It looks obvious that the binary exponentiation algorithm is more efficient than the naïve 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
```

Therfore, 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 mutiplications 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 explicitely 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 explicitely named for further use in the rest of the computation.

2.5.1 Addition chains

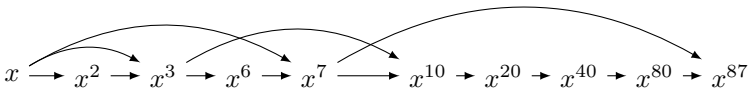
An *addition chain* (In short : *a chain*) [6] is a representation of a sequence of intermediate steps that lead to the evaluation of some x^n, under the assumption that each of these steps is a comutation of a power x^i, with $i < n$.

In articles from the combinatorician community, *e.g.* [6, 4] 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 two following sequences are addition chains for the exponent 87:

$$c_{87} = (1, 2, 3, 6, 7, 10, 20, 40, 80, 87) \tag{2.9}$$
$$c'_{87} = (1, 2, 3, 4, 7, 8, 16, 23, 32, 64, 87) \tag{2.10}$$

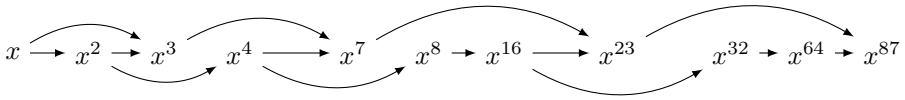
It is possible to associate to any addition chain a directed acyclic graph: whenever $i = j + k$, there is an arc from x^j to x^i and an arc from x^k to x^i. Figures 2.1 and 2.2 show the gphical representations of c_{87} and c'_{87}.

Figure 2.1: 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 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

Figure 2.2: Graphical representation of c'_{87} (10 multiplications)



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 [7, 8, 12], additions 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.

2.5.2 A type for addition chains

Let us recall that we want to represent some algorithms of the form described in section 2.5, but avoiding to represent intermediate results by **let-in** constructs. We describe below the main design choices we made:

- Continuation Passing Style (CPS) [33] 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) [16], the local variables associated to intermediate results are represented by variables of type A, where A is the underlying type of the considered monoid.

2.5.2.1 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 contruct (`Mult` *x y k*) is *"multiply* `x` *with* `y`*, then send the result of this multiplication to the continuation* `k`*"*.

From File ../V8.9/Powers/Chains.v

```
Inductive computation {A:Type}  : Type :=
| Return (a : A)
| Mult (x y : A) (k : A -> computation).
```

2.5.2.2 Monadic Notation

The following *monadic* notation makes terms of type `computation` look like expressions of a small programming language dedicated to sequences of mutiplications. Please look at *CPDT* [17] 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 2.3 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.
```

2.5.2.3 Definition

Thanks to the `computation` type family, we can associate a type to the kind of computation schemes described in Figures 2.1 and 2.2.

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`:

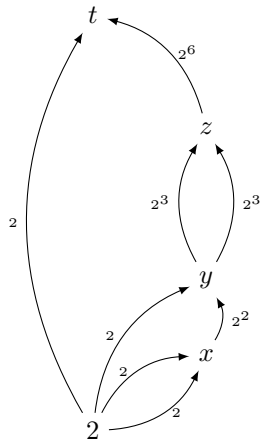


Figure 2.3: The dag associated to a computation of 2^7

```
Definition chain := forall A:Type, A -> @computation A.
```

Thus, terms of type `chain` describe polymorphic exponentiation algorithms.

For instance, Fig 2.4 shows a definition of the chain of Figure 2.1, for the exponent 87. Note that, like in PHOAS, bound variables associated with the intermediary results are Coq variables of type A.

```
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 2.4: A chain for raising x to its 87-th power

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 computations. 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
```

2.5.3 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 2.1 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.
```

```
 =  3232579099291745342922739807213602718533877%Z
     : Z
```

```
Compute chain_apply C87 (M:=M2N) (Build_M2 1 1 1 0)%N.
```

```
  = {|
      c00 := 1100087778366101931%N;
      c01 := 679891637638612258%N;
      c10 := 679891637638612258%N;
      c11 := 420196140727489673%N |}
     : M2 N
```

2.5.3.1 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 (c: chain) (n:nat) :=
  n <> 0 /\
  forall `(M:@EMonoid  A E_op E_one E_eq) (x:A),
      chain_apply c x ==   x ^ n.

Definition chain_correct (c: chain) (p:positive) :=
  chain_correct_nat c (Pos.to_nat p).
```

Definition 2.1 *A chain c is* optimal *for a given exponent p if its length is less or equal than 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.
```

2.6 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 2.5.3.1 on the previous 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. 2.7 on page 52.

2.6.1 Proof by rewriting

Let us show how to prove the correctness of some chains, using the `EMonoid` laws shown in Sect. 2.3.5 on page 27.

```
Ltac slow_chain_correct_tac :=
  match goal with
      [ |- chain_correct ?c ?p ] =>
      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 C7 7.
Proof.
   slow_chain_correct_tac.
Qed.
```

Unfortunately, this approach is terribly inefficient, even for quite small exponents:

```
Example C87_ok : chain_correct C87 87.
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 * (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.

2.6.2 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 contructors 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[5] and the numerous examples (including the `ring` tactic) in Coq's reference manual.

2.6.2.1 How does reflection work

Let us consider again the subgoal on page 43, 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 43,
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.

2.6.2.2 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.
```

2.6.2.3 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 *)
```

2.6.2.4 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. 2.3.1), 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)))
```

2.6.3 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.
```

2.6.3.1 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 naïve exponentiation definition.

```
Ltac reflection_correct_tac :=
match goal with
[ |- chain_correct ?c ?n ] =>
 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.
```

2.6.3.2 Example

The following dialogue clearly shows the efficiency gain over naïve setoid rewriting.

```
Example C87_ok : chain_correct C87 87.
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.

2.6.4 Chain correctness for —practically — free!

2.6.4.1 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. Stra-

chey [38] explores the nature of *parametric polymorphism*: "*Polymorphic functions behave uniformly for all types*" then Reynolds [32] formalizes this notion through binary relations. Wadler [40], then Cohen *et al.* [19] 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`.

2.6.4.2 Definition

A chain c is *parametric* if it has the same behaviour 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).
```

2.6.4.3 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 c  (the_exponent_nat 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 [19]). The reader may also consult a survey by D. Brown on the most important contributions to the notion of parametricity [9].

```
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 c (the_exponent 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).

2.6.4.4 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.
```

2.6.4.5 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 ?c ?p ] =>
apply param_correctness; parametric_tac
end.

Lemma C87_ok' : chain_correct C87 87.
Time param_chain_correct.
```

```
Finished transaction in 0.005 secs (0.005u,0.s) (successful)
```

```
Qed.
```

2.6.4.6 Remark

For the reasons exposed in Section 2.6.4.1 on page 48, 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.

2.7 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.

2.7.1 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 p (g 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. 2.6: 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 or equal than any chain returned by any correct generator:

```

Definition optimal_generator (g : positive -> chain) :=
 forall p:positive, optimal p (g p).
```

2.7.2 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
```

2.7.2.1 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.
```

2.7.2.2 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 2.5.2.3, and that this chain is correct (Sect 2.6.4.5 on page 51). 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.
```

2.7.2.3 Exercise

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.

2.8 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 [7] 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 $3n$.

2.8.0.0.1 Note: All the Coq material described in this section is available on File coq/Exponentiation/Euclidean_Chains.v

2.8.1 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 $3 \times n$.

Since `computation` is an inductive family of types, it could be possible to define a composition operator that works like list appending (replacing the `Return y` of the first computation with the second computation). *This approach is left as an exercise.* The solution we present is based on functional programming and the concept of continuation.

Exercice 2.1 Develop the approach suggested in the previous paragraph.

2.8.1.1 Type definition of f-chains

Let us consider *incomplete* or *open* chains. Such an object waits for another chain to resume a computation.

Figure 2.5 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.

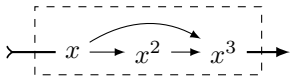


Figure 2.5: Graphical representation of `F3`

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.
```

2.8.1.2 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.
```

2.8.1.3 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 2.6 on the facing page).

```
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.
```

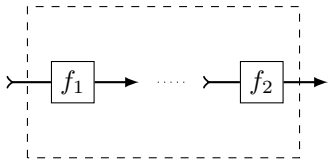


Figure 2.6: Composition of f-chains f_1 and f_2 (`Fcompose`)

2.8.1.4 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 (F2C F9) 9.
Proof.
  apply param_correctness_pos;  lazy;  parametric_tac.
Qed.
```

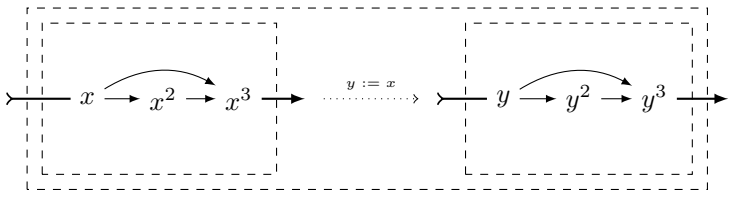


Figure 2.7: Composition of F-chains: F9

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
```

2.8.2 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  (f : Fchain) (n:nat) :=
  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 F3 3.
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.

2.8.2.0.1 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 (f : Fchain) (n:nat) :=
 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 (f : Fchain) (p:positive) :=
 Fchain_correct_nat f (Pos.to_nat p).
```

2.8.2.1 Examples

Let us show some manual correctness proofs for small f-chains:

```
Lemma F1_correct : Fchain_correct F1 1.
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 F3 3.
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 F2 2.
Proof.
  intros until M; intros k Hk a; simpl;
  apply Hk;  monoid_simpl M;  reflexivity.
Qed.
```

2.8.2.2 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 f1 n1 ->
                      Fchain_correct f2 n2 ->
```

```
                      Fchain_correct (Fcompose f1 f2)
                                     (n1 * n2).

(* 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. 2.8.3 on page 66) 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.
```

2.8.2.3 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).
```

2.8.2.4 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 (Fexp2_of_nat n) (2  ^ 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 (Fexp2 p) (2  ^ 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 f 144 /\
                                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
```

2.8.3 Building chains for two distinct exponents : k-chains

2.8.3.1 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 duplications 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 chosed to 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[2].

```
Definition Kkont A:=  A -> A -> @computation A.

Definition Kchain :=  forall A, Kkont A -> A -> @computation A.
```

2.8.3.2 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.
```

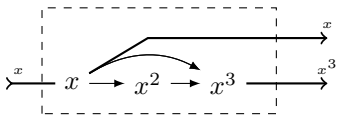


Figure 2.8: Graphical representation of `K3_1`

[2]The name `Kchain` comes from previous versions of this development. It may be changed later.

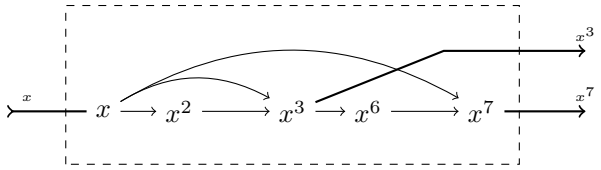


Figure 2.9: Graphical representation of `K7_3`

2.8.3.3 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 (kc : Kchain) (n p : nat) :=
  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 (kc : Kchain) (n p : positive) :=
  Kchain_correct_nat kc (Pos.to_nat n) (Pos.to_nat p).

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).
```

2.8.3.4 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 k7_3 7 3.
Proof.
  intros until M; intros; red; unfold k7_3; simpl.
  apply H; monoid_simpl M;  reflexivity.
Qed.
```

2.8.4 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 correcness 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 2.10 shows how `K7_3` is built as a composition of `K3_1` and `F2`.

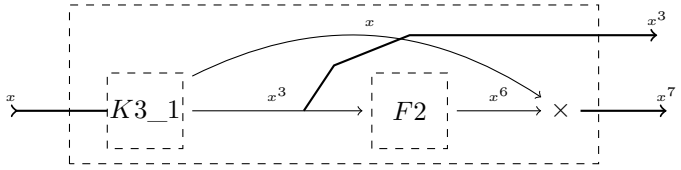


Figure 2.10: Decomposition of `K7_3`

2.8.4.1 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.

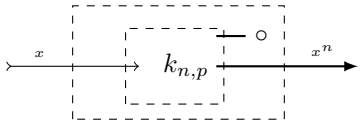


Figure 2.11: The `K2F (knp)` construction

```
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 knp) n.
(* Proof omitted *)

Instance K2F_proper (kc : Kchain)(_ : Kchain_proper kc) :
                                    Fchain_proper (K2F kc).

(* Proof omitted} *)
```

2.8.4.2 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.

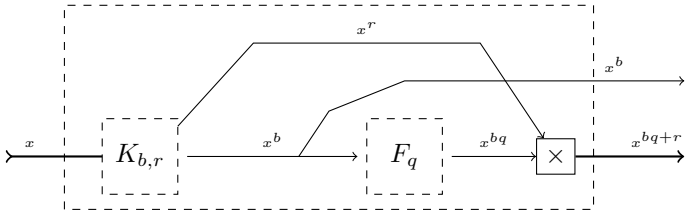


Figure 2.12: The KFK combinator

```
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 kbr b r ->
    Fchain_correct fq q ->
    Kchain_proper kbr ->
    Fchain_proper fq ->
    Kchain_correct (KFK kbr fq) (b * q + r) b.
(* Proof omitted *)

Instance KFK_proper :
forall (kbr : Kchain) (fq : Fchain),
  Kchain_proper kbr ->
  Fchain_proper fq ->
```

```
  Kchain_proper (KFK kbr fq)
(* Proof omitted *)
```

2.8.4.3 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.

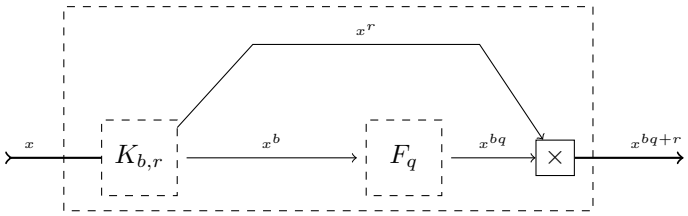


Figure 2.13: The KFF combinator

```
Definition KFF (kbr : Kchain) (fq : Fchain) : Fchain :=
  K2F (KFK kbr fq).

Lemma KFF_correct :
forall (b q r : positive) (kbr : Kchain) (fq : Fchain),
Kchain_correct kbr b r ->
Fchain_correct fq q ->
Kchain_proper kbr ->
Fchain_proper fq -> Fchain_correct (KFF kbr fq) (b * q + r).
(* Proof omitted *)


Instance KFF_proper :
forall (kbr : Kchain) (fq : Fchain),
Kchain_proper kbr -> Fchain_proper fq -> Fchain_proper (KFF kbr fq).
(* Proof omitted *)
```

2.8.4.4 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  Fp p ->
```

```
                    Fchain_proper Fp ->
                    Kchain_correct (FK Fp) p 1.
(* Proof omitted *)

Instance  FK_proper  (Fp : Fchain) (_ : Fchain_proper Fp):
  Kchain_proper (FK Fp).
(* Proof omitted *)
```

2.8.4.5 Computing x^p *and* x^{pq}

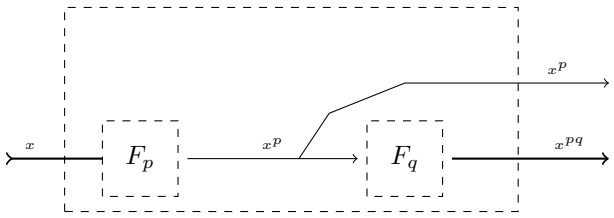


Figure 2.14: The FFK combinator

```
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 :
  forall (p q  : positive) (fp fq : Fchain),
    Fchain_correct fp p  ->
    Fchain_correct fq q ->
    Fchain_proper fp ->
    Fchain_proper fq -> Kchain_correct (FFK fp fq) (p * q ) p.
(* Proof omitted *)

Instance FFK_proper
   (fp: Fchain) (fq : Fchain)
    (_ :   Fchain_proper fp)
    (_ :  Fchain_proper fq) :  Kchain_proper (FFK fp fq) .
(* Proof omitted *)
```

2.8.4.6 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 F87 87.
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.

2.8.5 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 (make_chain p) 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 chosed 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 meaningfull only if $p < n$. Thus, in order to avoid a clumsy dependent pattern-matching, we chosed 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 explicitely 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 ch p
  | gen_K p d   => fun ch => Kchain_correct ch (p + d) p
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.
```

2.8.6 The dichotomic strategy

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 [7], 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 [7], the function that maps n into b is called a *strategy*. In this chapter, we concentrate on the so-called *dichotomic strategy*.

$\delta(n) = n \div 2^k$ 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.

Module teaser.Powers.Dichotomy contains a definition of the function `delta`, and proofs that if $n > 3$ then $1 < \delta(n) < n$.

2.8.7 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 several proof obligations, for proving that the measure on signatures is strictly decreasing along recursive calls. Thus, there are 9 such obligations, which take the form of inequalities between expressions that contain Euclidean divisions on positive numbers.

```
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
```

2.8.7.1 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 or proving properties of Euclidean division and binary logarithm. We plan to develop tools for allowing us to write shorter proof scripts.

```
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 (make_chain p) p.
Proof.
 intro p; destruct (chain_gen_OK (gen_F p)).
 unfold make_chain; apply F2C_correct; apply H.
Qed.
```

2.8.7.2 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
```

2.9 Projects

Project 2.2 (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 or equal than $\lfloor \log_2 n \rfloor$.

3. Prove that, for any positive n, the length of any Euclidean chain generated by the dichotomic strategy is always less or equal than 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 litterature (for instance [7].) 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*).

2.9.0.0.1 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.

Project 2.3 (Proof techniques)

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 II

Hydras and Ordinal Numbers

Chapter 3

Hydras and Hydra Games

3.1 Introduction

Hydra games appeared in an article published in 1982 by two mathematicians: L. Kirby and J. Paris [28]: *Accessible Independence Results for Peano Arithmetic, Kirby and Paris*. Although the mathematical contents of this paper are quite advanced, hydra games (a.k.a. *hydra battles*) are very easy to understand. There are now several sites on Internet where you can find tutorials on hydra games, together with simulators you can play with. See, for instance, the page written by Andrej Bauer [3]. 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.

Hydra battles, as well as Goodstein Sequences [25, 28] 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 [20], N. Dershowitz and G. Moser give a thorough survey on this topic [22].

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

- Representation in Coq of hydras and hydra battles
- A proof that every battle is finite and won by Hercules. This proof is based on a *variant* which maps any hydra to an ordinal strictly less than ϵ_0 and is strictly decreasing along any battle.
- Using a combinatorial toolkit designed by J. Ketonen and R. Solovay [27], we prove that, for any ordinal $\mu < \epsilon_0$, there exists no such variant mapping any hydra to an ordinal stricly less than μ. Thus, the complexity of ϵ_0 is realy needed for the previous proof.

We hope that such a work, besides exploring a nice piece of discrete maths, will show how Coq and its standard library are well fitted to help us to understand some non-trivial mathematical developments, and also to experiment the constructive parts of the proof through functional programming.

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

3.1.1 Remarks

In [28], Kirby and Paris showed that there is no proof of termination of all hydra battles in Peano Arithmetic (PA). Since we are used to writing proofs in higher order logic, the restriction to PA was quite unnatural for us. Thus, we chose to consider a class of proofs using the full expressive power of CIC, and to measure the difficulty of proving termination through an ordinal number.

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

Except in the `Schutte` library, dedicated to an axiomatic presentation of the set of countable ordinal numbers, all our development is axiom-free, and respects the rules of intuitionistic logic.

3.1.1.0.1 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 [28]
- "Rapidly growing Ramsey Functions" by Jussi Ketonen and Robert Solovay [27]
- "The Termite and the Tower", by Will Sladek [35]

3.2 On hydras

Technically, a *hydra* is just a finite ordered tree, each node of which has any number of sons. Note that, contrary to the computer science tradition, we will show the hydras with the heads up and the foot (i.e. the root of the tree) down. Fig. 3.1 represents such a hydra, which will be referred to as `Hy` in our examples (please look at the module `../V8.9/Ordinals/Hydra/Hydra_Examples.v`).

We use a specific vocabulary for talking about hydras. Table 3.1 shows the correspondance between our terminology and the usual vocabulary for trees in computer science.

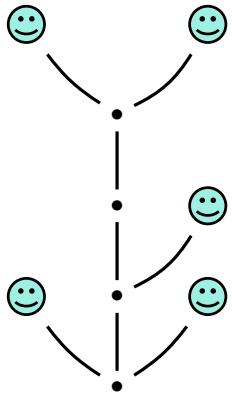


Figure 3.1: The hydra Hy

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

Table 3.1: Translation from hydras to trees

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.

3.2.1 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
  - 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 number* 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 chose its number of replication at every round of the battle[2].

Note that the description given in [28] of the replication process in hydra battles is also semi-formal.

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

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

3.2.2 Example

Let us start a battle between Hercules and the hydra `Hy` of Fig. 3.1.

At the first round, Hercules choses to chop off the rightmost head of `Hy`. Since this head is near the floor, the hydra loses this head. Let us call `Hy'` the resulting state of the hydra, represented in Fig. 3.2 on the facing page.

Next, assume Hercules choses to chop off one of the two highest heads of `Hy'`, for instance the rightmost one. Fig. 3.3 on the next page represents the rotten neck in dashed lines, and the part that will be replicated in red. Assume also that the hydra decides to add 4 copies of the red part. We obtain a new state `Hy''` depicted in Fig. 3.4.

Figs. 3.5 and 3.6 on page 88 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.

---

[1] h' will be called "the wounded part of the hydra" in the subsequent text. In Figures 3.3 on the facing page and 3.5 on page 88, 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.

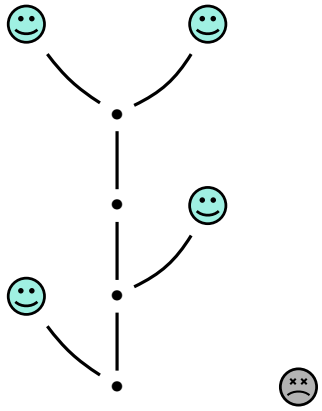


Figure 3.2: Hy': the state of Hy after one round

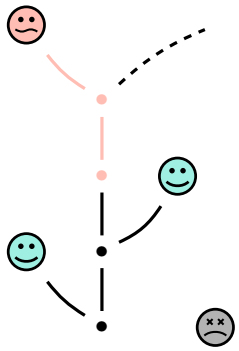


Figure 3.3: A beheading

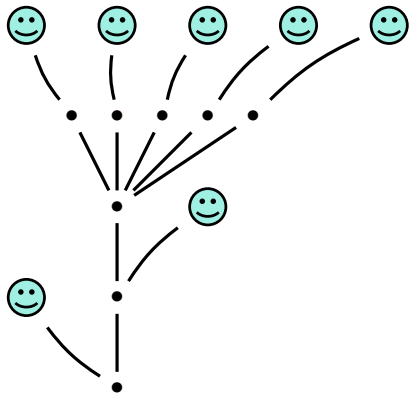


Figure 3.4: Hy", the state of Hy after two rounds

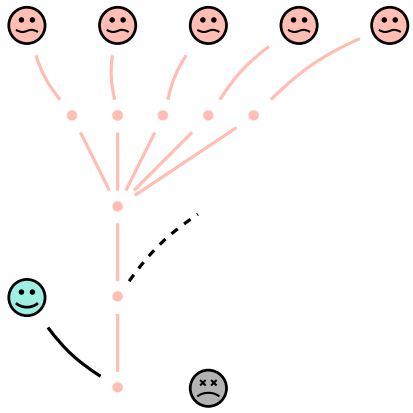


Figure 3.5: A third beheading (wounded part in red)

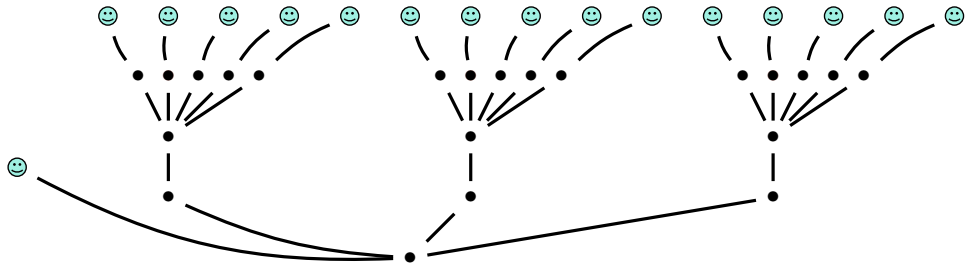


Figure 3.6: The configuration Hy”’ of Hy

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

3.3 Hydras and their representation in *Coq*

Various *Coq* developments represent trees the nodes of which have a non-fixed number of sons. The reader can, for instance, look at Chapter 14, pages 400-406 of [5]. Our representation uses two mutual inductive types: `Hydra` to represent any hydra, and `Hydrae` to represent finite sequences of hydras.

From ../V8.9/Ordinals/Hydra/Hydra_Definitions.v

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

Project 3.1 (**) 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 [5] or the archives of the Coq-club [21].

Project 3.2 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.

3.3.0.1 Abbreviations

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

From ../V8.9/Ordinals/Hydra/Hydra_Definitions.v

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

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

For instance, the hydra Hy of Figure 3.1 on page 85 is defined in *Gallina* as follows:

From ../V8.9/Ordinals/Hydra/Hydra_Examples.v

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

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

From ../V8.9/Ordinals/Hydra/Hydra_Definitions.v

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

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

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

For instance, the hydra Hy'' of Fig 3.4 on page 87 can be defined in Coq as follows:

From ../V8.9/Ordinals/Hydra/Hydra_Examples.v

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

3.3.0.2 Recursive functions on type `Hydra`

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

From ../V8.9/Ordinals/Hydra/Hydra_Definitions.v

```
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.
```

This definition results in the creation of three reduction rules:

```
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
```

Exercice 3.1 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.

3.3.1 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. "

3.3.1.1 A failed attempt

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

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

Ler us start a simple proof by induction.

From ../V8.9/Ordinals/Hydra/Hydra_Examples.v

```
Module Bad.

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

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

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

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

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

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

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

```
Abort.

End Bad.
```

3.3.1.2 A principle for 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
```

3.3.1.3 A correct proof

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

From ../V8.9/Ordinals/Hydra/Hydra_Examples.v

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

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

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

1. The first subgoal is as follows:

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

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

2. The second subgoal is trivial:

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

```
reflexivity.
```

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

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

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

Exercice 3.2 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 explicitely `P0`, and solves trivial subgoals. Apply it for writing a shorter proof of `height_lt_size`.

Exercice 3.3 The principles `Hydra_rect2` and `Hydrae_rect2`, which allow to build terms of sort `Type`, allow to build directly functions on types `Hydra` and `Hydrae`. Please redefine the function `hsize` as an application of `Hydra_rect2`, and prove that your version is extensionnaly equal to the one of Sect. 3.3.0.2.

3.4 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 one head off 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).

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 several cases, under the form of a bunch of [mutually] inductive predicates over the types `Hydra` and `Hydrae`.

We decompose the relation associated with each round in two cases:

R1 The chopped off head was at distance 1 from the foot.

R2 The chopped off head was at a distance greater or equal than 2 from the foot.

3.4.1 Chopping off a head at distance 1 from the foot (relation R1)

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

From ../V8.9/Ordinals/Hydra/Hydra_Definitions.v

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

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

3.4.1.1 Example

Let us represent in Coq the transformation of the hydra of Fig. 3.1 on page 85 into the configuration represented in Fig. 3.2.

From ../V8.9/Ordinals/Hydra/Hydra_Examples.v

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

3.4.2 Chopping of a head at distance ≥ 2 from the foot (relation R2)

Let us now consider beheadings where the chopped off head is at distance greater or equal than 2 from the foot. All the following relations are parameterized by the number n of new copies added by the hydra.

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 `R1 h h'` holds, in other words, `h'` is just `h`, without the chopped head. `S1` is an inductive relation with two constructors that allow to choose the position in s' of the wounded sub-hydra h.

From ../V8.9/Ordinals/Hydra/Hydra_Definitions.v

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

The rest of the definition is structured as two mutually inductive relations on hydras and sequences of hydras. The first constructor of `R2` describes the

case where the chopped head is exactly at height 2. The others constructors allow us to consider beheadings at height strictly greater than 2.

From ../V8.9/Ordinals/Hydra/Hydra_Definitions.v

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

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

3.4.2.1 Example

Let us prove the transformation of `Hy'` into `Hy''` (see Fig. 3.4 on page 87).

From ../V8.9/Ordinals/Hydra/Hydra_Examples.v

```
Example R2_example:  R2 4 Hy' Hy''.
Proof.
  right; right; left;  right; left; left; left; split; left.
Qed.
```

3.4.3 Description of a round

We combine the two cases above into one 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 ../V8.9/Ordinals/Hydra/Hydra_Examples.v

```
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 3.3 Give a direct translation of Kirby and Paris's description of hydra battles (quoted on page 86) and prove that our relational description is consistent with theirs.

3.4.4 Rounds and battles

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

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

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

Exercice 3.4 Prove the following lemma:

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

Remark 3.1 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.

Exercice 3.5 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.

Exercice 3.6 (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. 3.6 on page 88).

Hint: 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'`.

3.4.5 Classes of battles

In some presentations of hydra battles, e.g. [28, 3], 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 ../V8.9/Ordinals/Hydra/Hydra_Definitions.v

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

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

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

From ../V8.9/Ordinals/Hydra/Hydra_Definitions.v

```
Instance free : Battle.
Proof.
  refine (Build_Battle (fun i h h' => round h h') _); auto.
Defined
```

The `standard` class corresponds to an arithmetic progression of the replication factor : $0, 1, 2, 3, \ldots$.

From ../V8.9/Ordinals/Hydra/Hydra_Definitions.v

```
Instance standard : Battle.
Proof.
 refine (Build_Battle round_n _).
 intros i h h' H; now exists i.
Defined.
```

3.4.6 Fights

Let b be some 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 ../V8.9/Ordinals/Hydra/Hydra_Definitions.v

```
Inductive fight (B:Battle) : nat -> Hydra -> nat -> Hydra -> Prop :=
|  fight_1 : forall i h  h', battle_r   B i  h h' ->
                             fight B i h (S i) h'
| fight_n : forall i h  j h' h'',  battle_r  B i h h''  ->
                                   fight B (S i) h'' j h'  ->
                                   fight B i h j h'.
```

Remark 3.2 The class `free` is strongly related with the transitive closure `round_plus`, as expressed by the following lemmas.

From ../V8.9/Ordinals/Hydra/Hydra_Lemmas.v

```
Lemma fight_free_equiv1 : forall i j h h',
            fight free i h j h' ->   h -+-> h'.

Lemma fight_free_equiv2 : forall h h',
    h -+-> h' ->
   forall i, exists j,  fight free i h j h'.
```

3.5 A long battle

In this section we show how long a hydra battle can last. The following example considers a very small hydra, shown on figure 3.7.

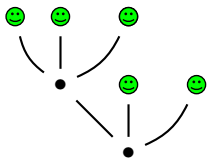


Figure 3.7: The hydra hinit

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

Moreover, in order to *compute* the length of a battle, we set Hercules strategy : to chop the rightmost head at each round, and the hydra's strategy : at the i-th round, the replication factor is exactly i (which means that the battle is a *standard* battle, see section 3.4.5 on the previous page).

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

We want to verify whether Hercules wins a standard battle starting from `hinit` at time $t = 0$, and in this case, in how many rounds. We assume Hercules' strategy is to chop off the rightmost head at every round.

All the experiment is described in file `../V8.9/Ordinals/Hydra/BigBattle.v`.

(* ICI *)

During the two first rounds, our hydra loses its two rightmost heads. Thus just before the third round, it looks like in figure 3.8.

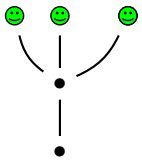


Figure 3.8: The hydra (hyd1 h3)

The following lemma is a formal statement about the two first rounds of the battle.

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

A first study with pencil and paper suggested us that, after three rounds, the hydra always look like in figure 3.9 on the next page (with a variable number of subtrees of height 1 or 0). Thus, we introduce handy notations.

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

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

For instance Fig 3.9 on the following page shows the hydra (`hyd 3 4 2`). The hydra `hyd 0 0 0` is the "final" hydra of any terminating battle, *i.e.* a tree whith exactly one node and no edge.

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

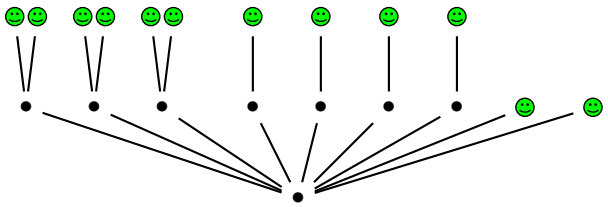


Figure 3.9: The hydra (hyd 3 4 2)

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

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

We would like to have a formal proof that the length of the considered battle is equal to some integer N, but we have first to *guess* this number. In order to study *experimentally* the different configurations of the battle, we will use a simple datatype for representing the states as tuples composed of the round number, and the respective number of daughters `h2`, `h1`, and heads of the current hydra.

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

The following function returns the next configurarion of the game. Note that this function is defined only for making experiments and is not "certified". Formal proofs about our battle will only start with the lemma `lemma:step-fight`, page 105.

```
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 => s
  end.
```

We can make bigger steps through iterations of `next`.

```
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 allows us to compute the state of the battle at round n.

```
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 computations above illustrate the limitations of blind tests. The battle we study seems to be awfully long. Let us concentrate our tests on some particular events : the states where `nh` $= 0$

From the value of `test 5`, it is obvious that at the 10-th round, the counter `nh` will be equal to zero.

```
 Compute test 10.
(*

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

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

```
 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
*)
```

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

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

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

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

```
Lemma step_fight : forall i a b c a' b' c',
   one_step i a b c a' b' c' ->
   fight 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 3.7 at time 0).

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

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

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

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

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

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

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

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

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

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

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

```
 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.
```

We are now able to make giant 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 3.10 represents the hydra at the M-th round. At the $M+1$-th round, it will look like in fig 3.11.

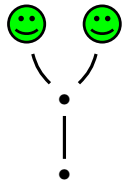


Figure 3.10
The state of the hydra after M rounds.

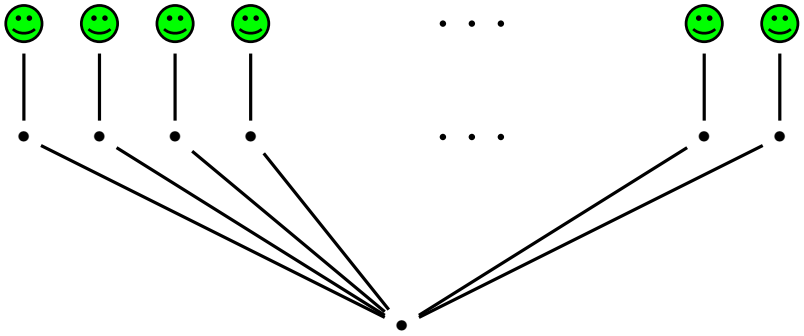


Figure 3.11
The state of the hydra after $M+1$ rounds (with $M+1$ heads).

```
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 fight is composed of N steps.

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

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

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

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

Using some facts (proven in `../V8.9/Ordinals/Hydra/BigBattle.v`) 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 or equal than $2^{2^{95}\times 95}$. If we wrote N in base 10, N would require at least 10^{30} digits!

3.6 Reasoning about any battle

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 [39] for instance).

3.6.1 Termination of all battles

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

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

3.6.2 Reachability

Let b be an instance of `battle`. We say that a configuration (i, h) is *reachable* if there exists some initial hydra h_0 such that the transformation from $(0, h_0)$ into (i, h) belongs to b. The following inductive definition considers also the case where $i = 0$ and $h = h_0$.

From ../V8.9/Ordinals/Hydra/Hydra_Definitions.v

```
Inductive reachable (b : Battle)
  : nat -> Hydra ->  Prop  :=
| reachable_0 : forall h1, reachable b 0 h1
| reachable_S : forall i h h0,  fight b  0 h0 i h ->
                                reachable b i h.
```

A nice property of the classes `free` and `standard` is that every configuration (i, h) is reachable. It is sufficient to add i heads to h's foot and start the battle at time 0.

From ../V8.9/Ordinals/Hydra/Hydra_Lemmas.v

```
Lemma reachable_free : forall i  h, reachable free i h.

Lemma reachable_standard  : forall h i, reachable standard i h.
```

3.6.3 Liveliness

We say that a kind b of battles is *alive* if for any reachable configuration (i, h), where h is not a head, there exists a further step in class b.

From ../V8.9/Ordinals/Hydra/Hydra_Definitions.v

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

The theorems `Alive_free` and `Alive_standard` of the module `../V8.9/Ordinals/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 ../V8.9/Ordinals/Hydra/Hydra_Lemmas.v

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

3.7 Termination

In this section, we are interested in proofs of termination of all battles for a given class of battles, and specifically proofs by *variants*.

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 $(1+, h')$ of a battle in b, the inequality $m(h') < m(h)$ holds.

From ../V8.9/Ordinals/Hydra/Hydra_Definitions.v

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

Exercice 3.7 Prove that, if there is an instance of `Hvariant Lt wf_Lt b m`, then there exists no infinite battle in `b`.

3.7.1 Some failed attempts

The class `Hvariant` is parameterized by a well-founded order $<$ on some type A, a class of battles b and a measure m. When $<$ or m are badly given for a given class b, one may fail to find a variant.

In this section, we present some naïve attempts, where we fail to build proof a termination of free battles.

3.7.1.1 Using the Hydra's height as a variant

A first plan to prove termination of all hydra battles is to use a simple measure that maps any hydra to a natural number. For instance, let us check whether the function *height* could be such a variant.

Figure 3.12

Unfortunately, that does not work. Please consider the hydras of Fig. 3.15 on page 116 and 3.12. The former can be transformed into the latter in one round, but their heights are equal.

From ../V8.9/Ordinals/Hydra/Omega_Small.v

```
Lemma height_bad :  ~ Hvariant lt_wf free height.
Proof.
  intros [H];
   specialize (H 1 (hyd1 (hyd2 head head))  (hyd1 (hyd1 head)));
   apply (lt_irrefl 2), H.
  -  apply reachable_free.
  -  discriminate.
  -  exists 0; right;  R2_here 0;  left.
Qed.
```

3.7.1.2 Using any variant defined on nat

We could imagine that our previous failure is due to the choice of `height` as a measure, and that a more complex function would work. In fact, we can prove that *no* instance of class `WfVariant round Peano.lt` m can be build, where m is *any* function of type `Hydra` $\rightarrow$ `nat`.

Let us present the main steps of that proof, the script of which is in the module Omega_Small.v [3].

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

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

We define an injection from the type `nat` into `Hydra`. For any natural number i, $\iota(i)$ is the hydra composed of a foot and $i+1$ heads at height 1. For instance, Fig. 3.13 represents the hydra $\iota(3)$.

[3] 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.

Figure 3.13: 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. 3.12 on the facing 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 replication factor.

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

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

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

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

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

More generally, we prove the following lemma:

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

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

- The base case $i = 0$ is trivial
- Otherwise, let i be any natural number and assume the inequality $i \leq m(\iota(i))$.

1. But the hydra $\iota(S(i))$ can be transformed in one round into $\iota(i)$ (by losing its righmost head, for instance)
2. Since m is a variant, we have $m(\iota(i)) < m(\iota(S(i)))$, hence $i < m(\iota(S(i)))$, which implies $S(i) \leq m(\iota(S(i)))$.

Then our proof is almost finished.

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

End Impossibility_Proof.
```

Exercice 3.8 Prove that there exists no variant m from `Hydra` into `nat` for proving the termination of all *standard* battles.

3.7.1.3 Lexicographic order on nat*nat

We prove now that even the type `nat * nat`, provided with the lexicographic product of `(nat,<)` by itself is too simple for proving the termination of all hydra battles. This impossibility result will prevent us from considering measures like the following one:

```
  Let m h = (height h, hsize h).
```

The proof we are going to develop has exactly the same structure as the previous one. Nevertheless, the proof of technical lemmas is a little more complex. In effect, the structure of the lexicographic order on $\mathbb{N} \times \mathbb{N}$ is more complex than the natural strict order $<$ on $\mathbb{N}$. Consider for instance that there exists an infinite number of pairs between $(1, 0)$ and $(2, 0)$.

Remark 3.3 The order structure we consider in this section is also known as the ordinal ω^2. We identify any pair $(i, j) \in \mathbb{N} \times \mathbb{N}$ with the ordinal $\omega \times i + j$. Thus the three kinds of ordinals in ω^2 are represented as follows:

null ordinal : the pair $(0, 0)$

successor ordinal : any pair (i, j) where $j > 0$

limit ordinal : any pair $(i, 0)$ where $i > 0$

The detailed proof script is in the file `../V8.9/Ordinals/Hydra/Omega2_Small.v`.

3.7.1.4 Preliminaries

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

From ../V8.9/Ordinals/Hydra/Omega2_Small.v

```
Section Impossibility_Proof.

Let t := (nat * nat)%type.

(** non-dependent lexicographic strict ordering on nat*nat *)

Let lt2 : relation t := lexico Peano.lt Peano.lt.

Infix "<" := lt2.

(** reflexive closure of lt2 *)
Let le2 := clos_refl _ lt2.
Infix "<=" := le2.

Variable m : Hydra -> t.

Context (Hvar : Hvariant lt2_wf free m).
```

Let us follow the same pattern as in Sect. 3.7.1.2. First, we define an injection from type `t` into `Hydra`. We associate with any pair (i, j) the hydra with i branches of length 2 and j branches of length 1.

From ../V8.9/Ordinals/Hydra/Omega2_Small.v

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

For instance, Figure 3.14 shows the hydra associated to the pair $(3, 5)$.

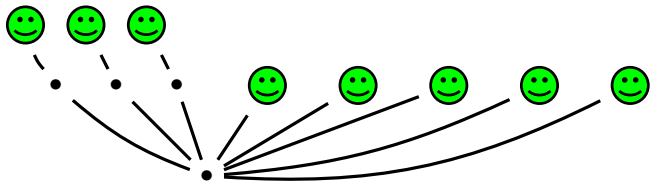


Figure 3.14: The hydra $\iota(3, 5)$

Like in Sect. 3.7.1.2, we build a hydra out of the range of `iota` (represented in Fig. 3.15 on the following page).

Figure 3.15
The hydra `big_h`.

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

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

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

Like in Sect. 3.7.1.2, we prove the double inequality `m big_h <= m small_h < m big_h`, which is impossible.

3.7.1.5 Proof of the inequality `m small_h < m big_h`

For proving the inequality `m_lt: m small_h < m big_h`, it suffices to build a fight 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` into the pair (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 folowing 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 < m big_h.
```

3.7.1.6 Proof of the inequality `m big_h <= m small_h`

The proof of the inequality `m big_h <= m small_h` is quite more complex than in Sect 3.7.1.2. If we consider some pair (i, j), where $i > 0$, there exists an infinite number of pairs stricly less than (i, j), and there exists an infinite number of battles that start from $\iota(i, j)$. In effect, at any configuration $\iota(k, 0)$, the hydra can freely chose 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 give more technical details.

- The proof of the lemma `m_ge : forall p : t, p <= m (iota p)` uses well-founded induction on p, and not structural induction on natural numbers
- For any pair p, we have to distinguish between three cases, according to the value of p's components.
  - $p = (0, 0)$
  - $p = (i, 0)$, where $i > 0$: p corresponds to a limit ordinal
  - $p = (i, j)$, where $j > 0$: p is the successor of $(i, j - 1)$.

Before starting the proof, we have to express the notion of *limit* in terms of least upper bounds, through the following logical equivalence.

From Omega2_Small.v.

```
Lemma limit_is_lub : forall i p,
   (forall j, (i,j) < p) <-> (S i, 0) <= p.
```

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

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

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

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

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

From ../V8.9/Ordinals/Hydra/Omega2_Small.v

```
Lemma m_ge : forall p : t,   p <= m (iota p).
Proof.
  intro p ; pattern p;
    apply  well_founded_induction with
               (R := lt2) (1:= wf_lexico lt_wf lt_wf);
     intros (i,j) IHij (* rest of proof skipped *)
```

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

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

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

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

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

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

3.7.1.7 End of the proof

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

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

From `../V8.9/Ordinals/Hydra/Omega2_Small.v`

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

End Impossibility_Proof.
```

Exercice 3.9 Prove that there exists no variant m from `Hydra` into `nat*nat` for proving the termination of all *standard* battles.
Hint: Make a copy of file `../V8.9/Ordinal/Hydra/Omega2_Small.v`, and replace `free` with `standard` in the declaration `Context (Hvar : Hvariant (Lt:=lt2) (wf_lexico lt_wf lt_wf) free m)`, then try to replay the proof.

Remark 3.4 In Chapter 5, we will prove a theorem that is much more general than the two previous results about $\mathbb{N}$ and $\mathbb{N} \times \mathbb{N}$. The proof of that general result will share the same structure, but will require a lot of technical results.

Logically, the two theorems we prove in this chapter are useless, because they are subsumed by a less specific one. Nevertheless, the proof structure is more apparent in these simple versions, which motivates their inclusion at this place of our document.

Exercice 3.10 Write *direct* proofs (*i.e.* without applying the result and tools of Chap. 5) that the following data structures are too simple for defining a variant for any hydra battle.

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

Chapter 4

A proof of termination

In this chapter, we present a formal and self contained proof of termination of all [free] hydra battles. First, we take from Manolios and Vroon [29] 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 an ordinal strictly less than ϵ_0.

4.1 Ordinal numbers

The proof of termination of all hydra battles presented in [28] is based on *ordinal numbers*. From a mathematical point of view, an ordinal is a representant of an equivalence class for isomorphims of strict, total and well-founded orders.

We can also associate to every ordinal α a set whose elements are all ordinals strictly less than α. Thus, it is meaningful to consider *finite*, *infinite*, *demunerable* and *non-countable* ordinals. The relation $<$ on ordinals is well-founded, and the order $\leq$ associated with $<$ is total.

We cannot cite all the litterature published on ordinals since Cantor's book [11], and leave it to the reader to explore the bibliography. Let us cite the book by Schütte [34] which contains an axiomatic definition of the set of countable ordinals we used as a mathematical specification of our implementaion in Coq [14].

Out of respect of the tradition, the meta-variables for ordinals will be α, β, γ, etc.

4.1.1 Definitions

Let α be an ordinal; we say that α is a *successor* if there exists some ordinal β such that α is is the least ordinal strictly greater than β.

We say that an ordinal λ is a *limit ordinal* is λ is the least upper bound of a stricly increasing sequence of ordinals. The meta-variable λ will be used for denoting limit ordinals.

An ordinal is either 0, a limit ordinal or a successor ordinal. This case analysis, as well as *transfinite* (*i.e.* well-founded) induction is used in many proofs about ordinal numbers.

The segment of finite ordinals is isomorphic to the set of natural numbers.

The first infinite ordinal is the limit ordinal ω.

The operations $+$, $\times$ and exponentiation on $\mathbb{N}$ are extended on ordinals numbers. Note that these extensions are not commutative any more. For instance $\omega = 1 + \omega \neq \omega + 1$ and $\omega = 2 \times \omega \neq \omega \times 2$. The ordinal ϵ_0 is the least solution of the equation $\alpha = \omega^\alpha$.

Please note that the set of countable ordinals is not countable.

Unless otherwise specified, we will only consider ordinals less than ϵ_0 in this chapter.

4.1.2 Ordinal Notations

It a proof assistant like Coq it may be useful to represent ordinals through some data-type, and make arithmetical operations and comparison effectively implemented through certified functions. Our user contribution [14] represents the set of ordinals less than ϵ_0 in Cantor normal form, and the set of ordinals less than Γ_0 in Veblen normal form.

4.2 The ordinal ϵ_0

The ordinal ϵ_0 is the least ordinal number that satisfies the equation $\alpha = \omega^\alpha$, where ω is the least limit ordinal. Thus, we can consider ϵ_0 as an *infinite* ω-tower.

Any ordinal less that ϵ_0 can be 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 a type for representing in `Coq` all the ordinals 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 compatible with the arithmetic operations of exponentiation of base ω and addition.

4.2.0.0.1 Remark Unless explicitly mentionned, the term "ordinal" will be used instead of "ordinal less than ϵ_0".

4.2.0.0.2 Remark One could think that it is useless to prove in Coq well known facts about ordinal numbers. Please keep in mind that we provide a *data structure* and functions written in Gallina for representing ordinals less than ϵ_0. Thus, we have to prove our representation is are correct w.r.t. the "classic" mathematical concepts. At least, we must prove the properties of the ordinal ϵ_0 that are really used in the proof of properties of hydra battles.

4.2.1 A data type for representing Cantor normal forms

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 ../V8.9/Ordinals/Epsilon0/T1.v

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

Remark

The name `T1` we gave to this data-type is proper to this development and refers to a hierarchy of ordinal notations. For instance we denoted the type of ordinals less than Γ_0 in Veblen normal form by an inductive type `T2`.

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

Another useful ordinal notation could be used for denoting the ordinals strictly less than ω^ω, but `list nat` could be used as well.

```
Inductive T0 : Set :=
  zero : T0
| cons : nat -> T0 -> T0.
```

4.2.1.1 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)).
```

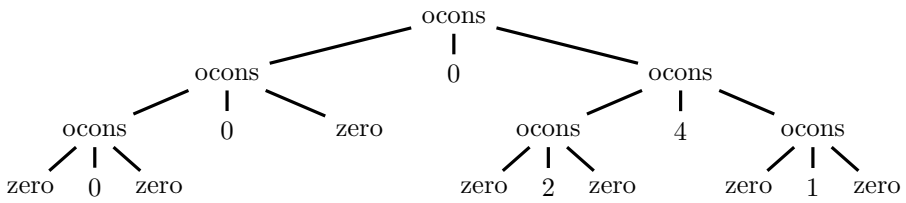


Figure 4.2: The tree-like representation of the ordinal $\omega^\omega + \omega^3 \times 5 + 2$

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

4.2.2 Abbreviations

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

4.2.2.1 Finite ordinals

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

```
Notation "'FS' n" :=
     (ocons zero n zero) (at level 29) : t1_scope.
```

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

Coercion fin  : nat >-> T1.
```

```
Example ten : T1 := 10.
```

4.2.2.2 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.

4.2.2.3 The ordinal ω^α, a.k.a. $\phi_0(\alpha)$

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

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

Remark 4.1 The name ϕ_0 comes from ordinal numbers theory. In [**?**], 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 6 on page 163.

4.2.2.4 The hierarchy of ω-towers:

The ordinal ϵ_0, although not represented by a finite term in Cantor normal form, is approximed by the sequance of ω-towers.

From ../V8.9/Ordinals/Epsilon0/T1.v

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

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

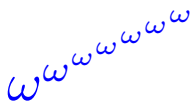


Figure 4.3: The ω-tower of height 7

4.2.3 Comparison between ordinal terms

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

From ../V8.9/Ordinals/Epsilon0/T1.v

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

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

From ../V8.9/Ordinals/Epsilon0/T1.v

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

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

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

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

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

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

From ../V8.9/Ordinals/Epsilon0/T1.v

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

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

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

From ../V8.9/Ordinals/Epsilon0/T1.v

```
Theorem lt_irrefl : forall alpha, ~ lt alpha  alpha.

Theorem lt_trans :
forall alpha beta: T1,
 lt alpha  beta ->
 forall gamma, lt beta gamma -> lt alpha gamma.

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

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

4.2.3.1 A Predicate for characterizing normal forms

We note that our data-type `T1` allows us to write expressions that are not properly in Cantor normal form as specified in Section 4.2. For instance, consider the following term of type `T1`.

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

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

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

From ../V8.9/Ordinals/Epsilon0/T1.v

```
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 bad_term.
```

```
   = false
     : bool
```

Remarks

We would like to get rid of terms of type `T1` which are not in Cantor normal form. In our development, we use indifferently three ways of specifying properties of proper ordinal terms.

- Prove statements of the form `forall alpha: T1, nf alpha -> P alpha`, where P is a predicate over type `T1`
- Define the restriction of some relations to terms in Cantor normal form:

  From ../V8.9/Ordinals/Prelude/Restriction.v

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

  From ../V8.9/Ordinals/Epsilon0/T1.v

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

  Definition LE := restrict nf le.
  Notation "alpha <= beta" := (LE alpha beta) : t1_scope.
  ```

  ```
  Example E0: ~ zero <= bad_term.
  Proof.
    compute; firstorder.
  Qed.

  Example E'0 : ~ bad_term <= zero.
  Proof.
  ```

```
  compute; firstorder.
Qed.
```

- Define a class `Ordinal`.

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

Definition lt (alpha beta : Ordinal) :=
   T1.LT (@cnf alpha) (@cnf beta).
Definition le (alpha beta : Ordinal) :=
   T1.LE (@cnf alpha) (@cnf beta).

Infix "<" := lt : epsilon0_scope.
Infix "<=" := le : epsilon0_scope.
```

Thus, theorem statements may have the following form: `forall alpha: Ordinal,` P `alpha`.

4.2.4 Syntactic definition of limit and successor ordinals

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

From ../V8.9/Ordinals/Epsilon0/T1.v

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

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

```
  Compute is_limit omega.
```

```
  = true
     : bool
```

```
Compute is_succ 42.
```

```
  = true
     : bool
```

The correctness of these definitions with respect to the mathematical notions of limit and successor ordinals is established through several lemmas. For instance, Lemma `canonS_limit`, page 147, shows that if α is (syntactically) a limit ordinal, then it is the least upper bound of a strictly increasing sequence of ordinals.

The following function allows us to discriminate three cases for any term of ype `T1`.

```
 Definition zero_succ_limit (alpha: T1) :
 {is_succ alpha} + {is_limit alpha} +  {alpha=zero}.
```

4.2.5 Arithmetic on ϵ_0

4.2.5.1 Successor

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

From ../V8.9/Ordinals/Epsilon0/T1.v

```
Fixpoint succ (alpha:T1) : T1 :=
 match alpha with zero => 1
            | ocons zero n _ => ocons zero (S n) zero
            | ocons beta n gamma => ocons beta n (succ gamma)
 end.
```

The following lemma establishes the connection between the functions `succ` and `is_succ`.

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

4.2.5.2 Addition and multiplication

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

```
Fixpoint plus (alpha beta : T1) : T1 :=
  match alpha,beta with
 |  zero, y  => y
 |  x, zero  => x
 |  ocons a n b, ocons a' n' b' =>
```

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

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

4.2.5.3 Examples

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

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

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

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

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

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

Lemma mult_not_monotonous :  exists alpha beta gamma : T1,
      alpha < beta /\ alpha * gamma = beta * gamma.
Proof.
```

```
  exists 3, 5, omega; now compute.
Qed.
```

4.2.6 Pretty printing Ordinals in Cantor normal Form

Let us consider again the ordinal α_0 defined in section 4.2.1.1 on page 124

If we ask Coq to print the normal form of `alpha_0`, we get a hardly readable term of type `T1`.

```
Compute alpha_0.
```

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

The function `pp: T1 -> ppT1` converts any closed term of type `T1` into a more readable expression.

```
Compute pp alpha_0.
```

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

4.3 Well-foundedness and transfinite induction

4.3.1 About well-foundedness

In order to use `T1` for proving termination results, we need to prove that our representation of ordinals less than ϵ_0 makes our order `<` 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 Contéjean [14]. That proof relies on a library on recursive path orderings written by E. Contéjean. We present also a direct proof of the same result, which does not require any knowledge on r.p.o.s.

Exercice 4.1 Prove that the *total* order `lt` on `T1` is not well-founded. **Hint:** You will have to build a counter-example with terms of type `T1` which are not in Cantor normal form.

4.3.1.1 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 less than β nor γ, and we cannot use the induction hypotheses on β nor γ.

```
Section First_attempt.

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

```
1 subgoal (ID 560)

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

```
  Abort.
```

The problem comes from that δ may be bigger that β or γ; for instance δ may be of the form `ocons` β' p' γ', where $\beta' \leq \beta$ and $p' < n$. Thus, the induction hypotheses `IHbeta` and `IHgamma` are useless for finishing our proof.

4.3.1.2 Using a stronger inductive predicate.

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

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

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

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

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

4.3.1.2.1 A helper First, we prove that, for any LT-accessible term α, any well formed term `ocons` α n β is also accessible:

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

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

```
  subgoal 1

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

Let `n:nat` and `beta:T1` such that `ocons alpha n beta` is in normal form. We prove first that `beta` is accessible, then we can 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 `../V8.9/Epsilon0/T1.v`

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

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

Theorem nf_Acc : forall alpha, nf alpha -> Acc LT  alpha.
Proof.
```

```
 induction alpha.
-  intro; apply Acc_zero.
 -  intros; eapply Acc_implies_Acc_strong;auto.
    apply IHalpha1;eauto.
    apply nf_inv1 in H; auto.
Defined.

Corollary T1_wf : well_founded LT.
```

And we have now our transfinite recursor:

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

Defined.
```

We are now able to define a tactic for doing transfinite induction on any ordinal $\alpha < \epsilon_0$.

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

Remark 4.2 The proof of well-foundedness using Évelyne Contejean's work on recursive path ordering is available in the module Epsilon0rpo.

4.4 A variant for hydra battles

4.4.1 Natural sum (a.k.a. Hessenberg's sum)

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

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

4.4.1.1 Definition of `oplus`

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

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

4.4.1.1.1 The 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`.

```
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 resulted in the quite complex form of `oplus`'s definition. For making easier proof of properties of this function, it is helpful to *derive* lemmas that will help to simplify expressions of the form `oplus a b`.

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

```
Lemma oplus_alpha_0 : forall alpha,  alpha o+ T1.zero = alpha.
Proof.
```

```
  destruct a; reflexivity.
Qed.
```

```
Lemma oplus_0_b : forall b, zero o+ b = b.
Proof.
  destruct b;reflexivity.
Qed.
```

```
Lemma oplus_eqn :
   forall a b,
     oplus a b =
     match a, b with
         zero, _ => b
       | _, zero => a
       | ocons a1 n1 b1, ocons a2 n2 b2 =>
         match compare  a1 a2 with
             Gt => ocons a1 n1 (oplus b1 b)
           | Eq => ocons a1 (S (n1 + n2)%nat) (oplus b1 b2)
           | Lt => ocons a2 n2 (oplus a b2)
         end
     end.
 Proof.
   destruct a, b; now cbn.
 Qed.
```

4.4.1.2.1 A hand-made induction principle *Coq* contains a command `Functional Scheme` that generates induction principles which correspond to recursive functions. Unfortunately, the current version (`8.9.1`) doesn't work on `oplus`, probably because of the inner `Fix`.

```
Functional Scheme oplus_ind := Induction for oplus Sort Prop.
```

```
Error: Anomaly "todo." Please report at http://coq.inria.fr/bugs/.
```

Fortunately, it's a good exercise for a semi-experienced user, to write her/himself induction principles similar to the ones returned by `Functional Scheme`.

- First, we chose to write a version for sort `Type`, since versions for sorts `Prop` and `Set` can be easily derived from the former one. According to *Coq*'s naming politics, we will call our principle `oplus_rect`
- The conclusion of `oplus_rect` will be `P a b (oplus a b)`, where `P` is an arbitrary function of type `T1 -> T1 -> T1 -> Type`
- The premises of `oplus_rect` will describe how to build an induction on the graph of `oplus`.

We are now ready to state and prove `oplus_rect`, and the reader will note that the statement is longer than the proof scrit itself, which is a standard proof by induction, simplification and case-analysis that follows `oplus`'s definition.

We associate also a tactic to the application of `oplus_rect`.

```
Lemma oplus_rect:
     forall P: T1 -> T1 -> T1 -> Type,
       (forall a:T1, P zero a a) ->
       (forall a: T1, P a zero a) ->
       (forall a1 n1 b1 a2 n2 b2 o,
          compare a1 a2 = Gt ->
          P b1 (ocons a2 n2 b2) o ->
          P (ocons a1 n1 b1) (ocons a2 n2 b2)
            (ocons a1 n1 o)) ->
       (forall a1 n1 b1 a2 n2 b2 o,
          compare a1 a2 = Lt ->
          P (ocons a1 n1 b1) b2 o ->
          P (ocons a1 n1 b1) (ocons a2 n2 b2)
          (ocons a2 n2 o)) ->
       (forall a1 n1 b1 a2 n2 b2 o,
          compare a1 a2 = Eq ->
          P b1 b2 o ->
         P (ocons a1 n1 b1) (ocons a2 n2 b2)
           (ocons a1 (S (n1 + n2)%nat) o)) ->
        forall a b, P a b (oplus a b).
Proof with auto.
   induction a.
   -    intro; simpl; destruct b;auto.
   -   induction b.
       + apply X0.
       + case_eq (compare a1 b1).
         * intro Comp; unfold oplus; rewrite Comp.
           cbn; apply X3 ...
         * intro Comp; cbn; rewrite Comp; apply X2...
         * intro Comp; cbn; rewrite Comp ...
 Defined.


Ltac oplus_induction a b:= pattern (oplus a b); apply oplus_rect.
```

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 Boundedness

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

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

Proof with auto.
  induction gamma; destruct alpha, beta.
(* Rest of proof omitted *)
```

4.4.2.2 Commutativity, associativity

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

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

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

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

Thus, we have just to apply the lemma **oplus_comm_0**.

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

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

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

```
    transfinite_induction alpha.
(* rest of proof omitted *)
```

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

4.4.2.3 Monotonicity

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

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

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

4.5 A variant for hydra battles

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

From ../V8.9/Ordinals/Hydra/Hydra_Termination.v

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

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

```
Theorem m_nf : forall h, nf (m h).
Proof.
 intro h; elim h using Hydra_rect2
```

```
            with (P0 := fun s =>  nf (ms s)).
 -  destruct h0; simpl; auto.
 -  constructor.
 - intros;  rewrite ms_eqn2; apply oplus_nf.
  + now apply nf_phi0.
  + assumption.
Qed.

Theorem ms_nf : forall s, nf (ms s).
Proof with auto with T1.
 induction s...
  rewrite ms_eqn2...
  apply oplus_nf...
  apply nf_phi0; now  apply m_nf.
Qed.
```

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' < ms s.
```

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

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

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

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

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

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

Theorem every_battle_terminates  : Termination.
```

```
Proof.
  red; apply Inclusion.wf_incl with
         (R2 := fun h h' =>  m h < 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 mapping `Hydra` into $[0, \epsilon_0[$ for proving the termination of any hydra battle
- There exists *no* such variant from `Hydra` into $[0, \omega^2[$ and 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" ?

A non-trivial variant of this question is the following one: " Does there exist any strictly decreasing variant mapping type `Hydra` into some interval $[0, \mu[$ (where $\mu < \epsilon_0$) for proving the termination of all *standard* hydra battles" ?

The next chapter is dedicated to the proof that both question have a negative answer.

Chapter 5

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 [28], 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 some segment of ordinals.

Let sum up the main results proved in the previous chapter, about the termination of all hydra battles.

- There is no variant mapping the type `Hydra` into the interval $[0, \omega^2[$ (section 3.7.1.3 on page 114), and a fortiori $[0, \omega[$
- There is a variant that maps the type `Hydra` into the interval $[0, \epsilon_0[$ (theorem `every_battle_terminates`, in section 4.5 on page 140).

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 ?"

In this chapter, we prove in Coq the following results:

> 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. They share much material with Kirby and Paris' , although they do not use any knowledge about Peano arithmetic nor model theory. It it written in plain CIC, and is mainly composed on function definitions and proofs of properties of these functions. Nevertheless, all the tools we use come from an article by J. Ketonen and R. Solovay [27], already cited in the work by L. Kirby et J. Paris on the termination of Goodstein sequences and hydra battles [28]. 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.

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 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 of $\{\lambda\}(i)$ $(i \in \mathbb{N}_1)$
- 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 $\iota(\{\alpha\}(i)$ in one round with the replication factor i
- From the two previous properties, we infer that whenever $\beta < \alpha < \epsilon_0$, there exists a (free) fight from $\iota(\alpha)$ to $\iota(\beta)$.

5.2.0.0.1 Remark In [27], canonical sequences are defined for any ordinal $\alpha < \epsilon_0$, by stating that if α is a successor ordinal, the sequence associated with α is simply the constant sequence whose terms are equal to the predecessor of α. Likewise, we define the canonical sequence of 0 as the sequence whose all terms are equal to 0.

This convention allows us to make total the function that maps any ordinal α and natural number i to the i-th item of the canonical sequence associated with α.

Firs, let us recall how canonical sequences are defined in [27]. For efficiency's, we decided not to implement directly K.&S's definitions, but to define in Gallina simply typed structurally recursive functions which the same abstract properties.

5.2.0.1 Mathematical definition of canonical sequences

In [27] 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 $\omega^\beta \times (\gamma + 1)$.

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**, etc., are available. For simplicity's sake, we used an auxiliary function `canonS` of type `nat -> T1 -> T1` such that `canonS i α` is equal to $\{\alpha\}(i+1)$.

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

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

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

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

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

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

```
Example canonseq3 :  canonseq 3 (omega ^ omega) = omega ^ 3.
Proof. reflexivity. Qed.
```

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

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

5.2.1 Basic properties of canonical sequences

We did not try to prove that our definition really implements Ketonen and Solovay's [27]'s canonical sequences. The most important is that we are able to prove the abstract properties of canonical sequences that are really used in our proof. The complete proofs are in the module `../V8.9/Ordinals/Epsilon0/KS.v`.

Our definition of function `canonS` makes the following verification trivial.

```
Lemma canonS_zero : forall i, canonS i zero = zero.
Proof. reflexivity. Qed.
```

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

```
Lemma canonS_succ i alpha :
  nf alpha ->  canonS i (T1.succ alpha) = alpha.
Proof.
 induction alpha.
 (* rest of proof omitted *)
```

5.2.1.1 Canonical sequences and the order $<$

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

From ../V8.9/Ordinals/Epsilon0/KS.v

```
Lemma canonS_LT : forall i alpha, nf alpha -> alpha <> T1.zero ->
                              (canonS i alpha <  alpha)%t1.
```

5.2.1.2 Limit ordinals are really limits

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

Note the use of Coq's `sig` type in the theorem's statement, which relates the boolean function `is_limit` 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.

From ../V8.9/Ordinals/Epsilon0/KS.v

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

Proof.
  transfinite_induction_LT lambda.
  (* rest of proof omitted *)
Defined.
```

```
Lemma canonS_limit : forall lambda, nf lambda -> is_limit lambda  ->
                                strict_lub (fun i => canonS i lambda) lambda.
```

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

5.2.1.3 Paths inside ϵ_0

Let us consider the transitive closure of the relation associated to the function `canonS`. We will call a *path* from α to β any sequence of *steps*, each step being a pair $(\alpha, \{\alpha\}(i))$ for some integer $i > 0$.

```
Definition small_step : relation T1 :=
  fun alpha beta =>  exists i,  beta = canonS i alpha.

Definition path := clos_trans_1n T1 small_step.
```

From the lemma `canonS_LT`, we convert any path into an inequality on ordinals (by induction on transitive closures).

From teaser.Ordinals.Epsilon0.KS

```
Lemma path_LT (alpha beta : T1) :
     path alpha beta ->
     beta  <> zero -> nf alpha -> nf beta ->
     beta < alpha.
  Proof.
    induction 1.
    (* rest of the proof skipped *)
```

The proof of the converse lemma is a little more complex: it is mainly a transfinite induction, using the lemma `canonS_limit_strong`. We advise the reader to replay the proof in order to better undersstand its structure and the use of the `sig` type in Coq.

```
Lemma LT_path (alpha beta : T1) :
  beta < alpha -> path alpha beta.
Proof.
  transfinite_induction alpha;
 (* rest of proof skipped *)
```

Let us look at the handling of limit ordinals in this proof.

```
beta, alpha : T1
IHalpha : forall y : T1, y < alpha -> beta < y -> path y beta
H : beta < alpha
H0 : nf beta
H1 : nf alpha
Hlimit : is_limit alpha
============================
path alpha beta
```

Since α is a limit, there exists some j such that $\beta < \{\alpha\}(j+1)$.

```
 destruct  (canonS_limit_strong H1 Hlimit H) as [j Hj];
           apply path_trans with (canonS j alpha).
```

The first subgoal is trivial.

```
2 subgoals (ID 187)

 beta, alpha : T1
  IHalpha : forall y : T1, y < alpha -> beta < y -> path y beta
  H : beta < alpha
  H0 : nf beta
  H1 : nf alpha
  Hlimit : is_limit alpha
  j : nat
  Hj : beta < canonS j alpha
  ============================
  path alpha (canonS j alpha)
```

```
left; now exists j.
```

The second subgoal is just an application of the induction hypothesis `IHalpha`.

```
 beta, alpha : T1
  IHalpha : forall y : T1, y < alpha ->
         beta < y -> path y beta
  H : beta < alpha
  H0 : nf beta
  H1 : nf alpha
  Hlimit : is_limit alpha
  j : nat
  Hj : beta < canonS j alpha
  ============================
  path (canonS j alpha) beta
```

```
apply IHalpha; auto.
       {  apply canonS_LT; auto.
          apply is_limit_not_zero;auto. }
```

Thus, canonical sequences are a way to decompose any inequality $\beta < \alpha < \epsilon_0$ into a finite sequence of elementary – successor or limit – steps.

5.2.2 Canonical sequences and hydra battles

In order to apply our knowledge about ordinal numbers (less than ϵ_0) to the sudy of hydra battles, we define an injection from the interval $[0, \epsilon_0[$ into the type `Hydra`.

```
Fixpoint iota (alpha : T1) : Hydra :=
  match alpha with
  | T1.zero => head
  | ocons gamma n beta => node (hcons_mult (iota gamma) (S n)
```

```
                                          (iotas beta))
  end
with iotas (alpha : T1) :  Hydrae :=
       match alpha with
       | T1.zero => hnil
       | ocons alpha0 n beta  =>
           hcons_mult (iota alpha0) (S n)
                              (iotas beta)
       end.
```

For instance Fig. 5.1 shows the image by ι of the ordinal $\omega^{\omega+2}+\omega^\omega\times 2+\omega+1$

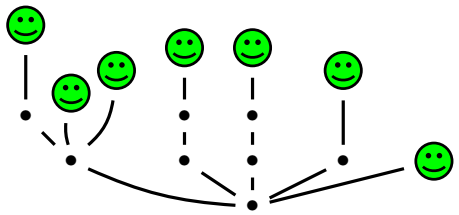


Figure 5.1: The hydra $\iota(\omega^{\omega+2}+\omega^\omega\times 2+\omega+1)$

The following lemma (proved in Ordinals.Hydra.O2H.v) maps the binary relation associated to canonical sequences to rounds of hydra battles.

```
Lemma canonS_iota i alpha :
    nf alpha -> alpha <> 0 ->
    iota alpha -1-> iota (canonS i alpha).
```

The next step of our development is to extend this relationship to the order $<$ on $[0,\epsilon_0[$ on one side, and hydra fights on the other side.

```
Lemma path_to_fight alpha beta :
     nf alpha -> nf beta -> alpha <> 0 ->
     path alpha beta -> iota alpha -+-> iota beta.
```

As a corollary, we are now able to transform any inequality $\beta<\alpha<\epsilon_0$ into a (free) fight.

```
Lemma LT_to_fight alpha beta :
    beta < alpha ->  iota alpha -+-> iota beta.
```

5.3 A first proof of impossibility

We have now got all the tools for proving there is 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 may be considered as a generalization of the proofs described in sections 3.7.1.2 on page 112 and 3.7.1.3 on page 114. We advise the reader to compare the three proofs step by step, lemma by lemma.

In the module Hydra.Epsilon0_Needed_Generic, we assume there exists some variant m bounded by some ordinal $\mu < \epsilon_0$. This part of the development is parameterized by some class B of battles, which will be instantiated later to `free` or `standard`.

```
Class BoundedVariant  (B:Battle) :=
  {
    mu:T1 ;
    m: Hydra -> T1;
    mu_nf : nf mu;
    Hvar : Hvariant T1_wf B m;
    m_bounded : forall h, m h < mu
  }.
```

Let us assume there exists such a variant:

```
Section Bounded.
  Context (B: Battle)
          (Hy : BoundedVariant B).
```

The following property is not provable for any instance of B : `Battle`. Nevertheless, it is satisfied by the instances `free` and `standard` of class `Battle`. In order to "factorize" some proofs about these instances, we assume this property as an hypothesis of the current section.

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

First, we prove by transfinite induction over α a minoration of the measure of the hydra $\iota(\alpha)$.

```
Lemma m_ge alpha:  nf alpha -> alpha <= m (iota alpha).
  Proof.
```

- 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 3.7.1.2 on page 112 and 3.7.1.3 on page 114 contain a similar lemma, also called `m_ge`. We are now able to build a counter-example.

```
Definition big_h := iota mu.
Definition beta_h := m big_h.
Definition small_h := iota beta_h.
```

From Lemma `m_ge` we infer the following inequality :

```
Corollary m_ge_generic : m big_h <= m small_h.
```

The (big) rest of the proof is dedicated to prove formally the converse inequality `m small_h < m big_h`.

5.3.1 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 Hydra.Epsilon0_Needed_Free. The case $B =$ `standard` will be studied in section 5.4 on the next page.

```
Section Impossibility_Proof.

Context (Var : BoundedVariant free ).
```

1. The following lemma is an application of `m_ge_generic`, since `free` satisfies trivially the hypothesis `m_decrease`.

```
Lemma m_ge : m big_h <= m small_h.
  Proof.
    apply m_ge_generic.
    intros;  generalize Hvar ;  destruct 1.
    apply variant_decr with i.
    intro ; subst; now apply (head_no_round_n _  _ H).
    exists i; apply H.
  Qed.
```

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

3. By Lemma `LT_to_fight`, we get a (free) battle from `big_h = iota mu` to `small_h = iota (m big_h)`.

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

4. From the hypotheses on m, we infer:

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

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

```
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
     : BoundedVariant free -> False
```

5.4 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_fight`, which builds a battle out of any inequality $\beta < \alpha$. This lemma is a straightforward application of `LT_path`: every approximant $\{\alpha\}(j+1)$ built when α is a limit ordinal gives a round with j as the replication factor. Since j depends on β, we cannot be sure that the generated battle is a genuine standard battle.

The tool we need to use is once again in Ketonen and Solovay's article [27]. Instead of considering plain paths, i.e. sequences $\alpha_0 = \alpha, \alpha_1, \ldots, \alpha_k = \beta$ where α_{j+1} is equal to $\{\alpha_j\}(i_j)$ for some i_j, we will consider various constraints on these sequences. Please note that the vocabulary on paths is ours, but all the concepts come really from [27].

Such 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. In Coq, standard paths can be defined as follows.

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

Inductive standard_pathR(j:nat)( beta:T1):  nat -> T1 -> Prop :=
  std_1 : forall i alpha, beta = canonseq i alpha -> j = S i ->
                                standard_pathR j beta i  alpha
```

```
| std_S : forall i alpha,
      standard_pathR j beta (S i) (canonseq i alpha)  ->
      standard_pathR j beta i alpha.


Definition standard_path  i alpha j beta :=
   standard_pathR j beta i alpha.
```

Inside the mathematical text and figures, we shall use the notation $\alpha \underset{i,j}{\longrightarrow} \beta$ for the proposition `standard_path` $i\ \alpha\ j\ \beta$.

In [27] the notation is $\alpha \overset{*}{\underset{i}{\to}} \beta$ for the proposition $\exists j, i < j \wedge \alpha \underset{i,j}{\longrightarrow} \beta$.

It would be nice to transform any inequality $\beta < \alpha < \epsilon_0$ into a standard path $\alpha \underset{0,j}{\longrightarrow} \beta$ for some j, then into a standard battle from $\iota(\alpha)$ to $\iota(\beta)$. Following [27], 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*, then to a standard path.

5.4.1 Paths with constant index

Happily, the aforementionned article contains a combinatorial study of paths. 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.

Like in [27], we shall use the notation $\alpha \underset{i}{\to} \beta$ for denoting such a path.

```
Definition const_pathS i :=
    clos_trans_1n T1 (fun alpha beta => beta = canonS i alpha).

Definition const_path i alpha beta :=
  match i with
    0 => False
  | S j => const_pathS j alpha beta
end.
```

Please note that the relation `const_path` is functional: given i, α and l, the following function returns the ordinal β such that there exists a path $\alpha \underset{i+1}{\longrightarrow} \beta$ of length l.

```
Fixpoint const_funS (i:nat)(alpha : T1)(l:nat):  T1  :=
  match l
  with
  | 0 => alpha
  | S m => const_funS i (canonS i alpha) m
  end.
```

The following computations show applications of `constS_fun` to the ordinal ω^ω, with various values of i and l.

```
Compute  (const_funS 2 (omega ^omega)  55).
```

```
  = zero
     : T1
```

```
Compute pp (const_funS 2 (omega ^omega) 15).
```

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

```
Compute pp (const_funS 4 (omega^omega)  100).
```

```
= (omega ^ 4 * 4 + omega ^ 3 * 4 + omega ^ 2 + omega * 4 + 4)%pT1
     : ppT1
```

A most interesting property of such paths is that we can "upgrade" their index, as stated by K.&S.'s Corollary 12.

```
Corollary C12 (alpha : T1) :  nf alpha ->
         forall beta i n, beta  < alpha  ->
                (i < n)%nat ->
                 const_pathS i alpha beta ->
                 const_pathS n alpha beta.
Proof.
  transfinite_induction_lt alpha.
  (* (long) proof skipped *)
```

We shall often use a version of `C12` with large inequalities.

```
Corollary C12' (alpha : T1) :  nf alpha ->
         forall beta i n, (beta < alpha)%t1 ->
               (i <= n)%nat ->
               const_pathS i alpha beta ->
               const_pathS n alpha beta.
```

5.4.1.1 Sketch of proof of `C12`

We prove this lemma by transfinite induction on α. Let us consider a path $\alpha \underset{i}{\to} \beta$ $(i > 0)$. Its first step is the pair $(\alpha, \{\alpha\}(i))$, We have $\{\alpha\}(i) < \alpha$ and $\{\alpha\}(i) \underset{i}{\to} \beta$. Let n be any natural number such that $n > i$. By the induction hypothesis, there exists a path $\{\alpha\}(n) \underset{i}{\to} \beta$.

- If α is a successor ordinal $\gamma + 1$, then $\{\alpha\}(n) = \{\alpha\}(i) = \gamma$. Thus we have a path $\alpha \underset{n}{\to} \gamma \underset{n}{\to} \beta$

- If α is a limit ordinal, we apply the following theorem (numbered `2.4` in Ketonen and Solovay's article).

```
Theorem Theorem_2_4 (lambda : T1) :
   nf lambda ->
   is_limit lambda  ->
   forall i j, (i < j)%nat ->
               const_pathS 0 (canonS j lambda)
                             (canonS i lambda).
```

  We build the following paths :

  1. $\alpha \xrightarrow{n} \{\alpha\}(n)$
  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 5.1 `C12` "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.

Exercice 5.2 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 `C12` so useful? Let us consider two ordinals $\beta < \alpha < \epsilon_0$. By induction on α, we decompose any inequality $\beta < \alpha$ into $\beta < \{\alpha\}(i) < \alpha$, where i is some integer. Applying collorary `C12'` 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 [27] is naturally expressed in terms of Coq's `sig` construct.

```
Lemma L2_6_1 (alpha : T1) :
  nf alpha ->
  forall beta,  beta < alpha  ->
                       {n:nat | const_pathS n alpha beta}.
Proof.
  transfinite_induction alpha.
  (* rest of proof skipped *)
```

Intuitively, lemma `L2_6_1` shows that if $\beta < \alpha < \epsilon_0$, then there exists a battle from $\iota(\alpha)$ to $\iota(\beta)$ where the replication factor is constant, although large enough.

5.4.2 Casting paths with constant index into standard paths

The article [27] contains the following lemma, the proof of which is quite complex, which allows to simulate i-paths by $[i+1, j]$-paths, where j is large enough.

```
(* Lemma 1 page 300 of [KS] *)

Lemma constant_to_standard_path
  (alpha beta : T1) (i : nat):
  nf alpha -> const_pathS i alpha beta -> zero  < alpha ->
  {l:nat | standard_path (S i) alpha j beta}.
```

5.4.2.1 Sketch of proof of `constant_to_standard_path`

Our proof follows the proof by Ketonen and Solovay, including its organization as a sequence of lemma. Since it is a non-trivial proof, we will comment its main steps below.

Préliminaries

Please note that, given an ordinal α : `T1`, and two natural numbers i and l, there exists at most a standard path $\alpha \xrightarrow[i,i+l]{*} \beta$. The following function computes β from α, i and l.

```
 Fixpoint standard_fun (i:nat)(alpha : T1)(l:nat):  T1  :=
  match l
  with
  | 0 => alpha
  | S m => standard_fun (S i) (d i alpha) m
  end.
```

```
  Compute standard_fun 2 omega 15.
(*   = zero
     : T1 *)
Compute pp (standard_fun 2 (omega^omega)  10).
(*
= (omega + 7)%pT1
     : ppT1
*)
Compute pp (standard_fun 4 (omega^omega)  100).
(*
 = (omega ^ 3 * 4 + omega ^ 2 * 5 + omega * 3 + 39)%pT1
     : ppT1 *)
```

By transfinite induction over α, one prove that the ordinal 0 is reachable from any ordinal $\alpha < \epsilon_0$ by some standard path.

```
Lemma standard_path_to_zero :
  forall  alpha i, nf alpha ->
                  {j: nat | standard_path (S i) alpha j zero}.
```

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 <  beta)
            (p : const_pathS n alpha  beta).
```

Applying `standard_path_to_zero`, 0 is reachable from α by some standard path (see figure 5.2).

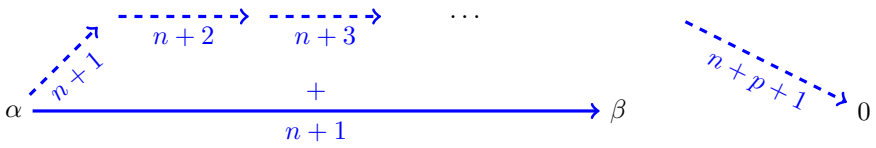


Figure 5.2: la belle-preuve (1)

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.

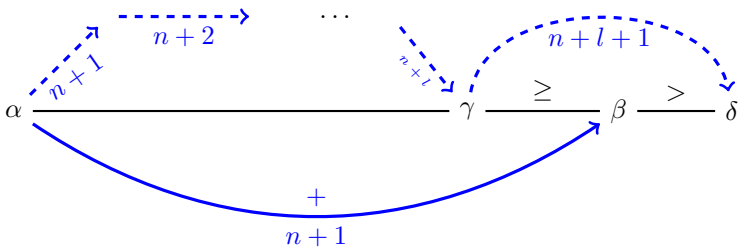


Figure 5.3: la belle preuve (2)

- 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 $\delta = \{\gamma\}(n+l+2)$. By applying several times lemma `C12`, one converts all paths into $n+l+1$-paths (see figure 5.4).

  But γ is on the $n+l+1$-path from α to β. As shown by figure 5.5), the ordinal δ, reachable from γ in one single step, must be greater or equal than β, which contradicts our hypothesis $\beta < \gamma$.

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

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

End Constant_to_standard_Proof.


Lemma constant_to_standard_path
  (alpha beta : T1) (i : nat):
  nf alpha -> const_pathS i alpha beta -> zero  < alpha ->
  {j:nat | standard_path (S i) alpha j beta}.
```

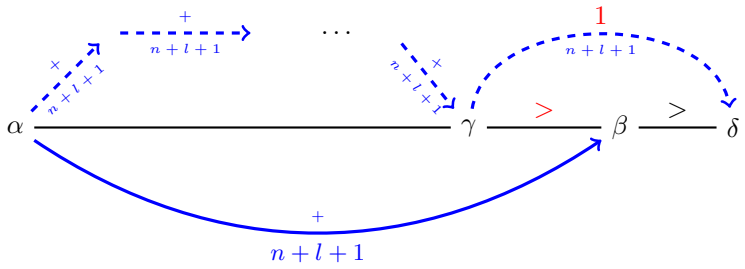


Figure 5.4: la belle preuve (3)

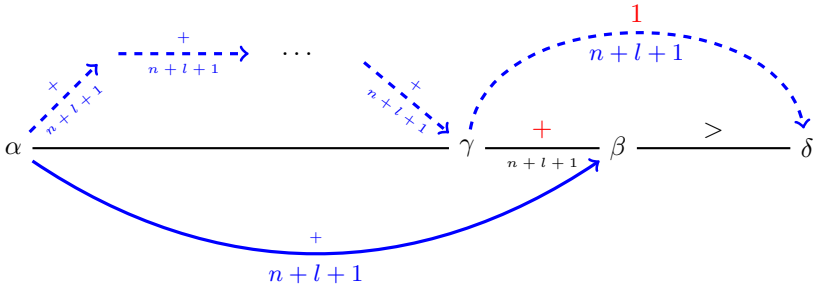


Figure 5.5: fin de la belle preuve

Applying `L2_6_1` and `constant_to_standard_path`, we get the following corollary.

```
Corollary  LT_to_standard_path
      (alpha beta : T1) :
  beta   < alpha ->
  {n : nat & {j:nat | standard_path (S n) alpha j beta}}.
```

5.4.3 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. The proof we are going to comment can be consulted in the module `../V8.9/html/teaser.Ordinals.Hydra.Epsilon0_Needed_Std.html`. Please note that it has the same global structure as in section5.3.1

Applying the lemmas `L2_6_1` of the module `L2_6_1` and `constant_to_standard_path`, we can convert any inequality $\beta < \alpha < \epsilon_0$ into a standard path from α to β, then into a fragment of a standard battle from $\iota(\alpha)$ to $\iota(\beta)$.

```
Lemma lt_to_standard_battle :
    forall alpha beta,
      beta < alpha ->
      exists n i,  fight standard  n (iota alpha) i (iota beta).
```

Next, please consider the following context:

```
Section Impossibility_Proof.

  Context (Var : BoundedVariant standard).
```

In the same way as for free battles, we import a large inequality from the module `../V8.9/html/teaser.Ordinals.Hydra.Epsilon0_Needed_Generic.html` .

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

If remains to prove the following strict inequality, in order to have a contradiction.

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

Proof: Let us recall that `big_h` $= \iota(\mu)$ and `small_h` $= \iota(m($`big_h`$))$.

Since $m($`big_h` $) < \mu$, there exists a standard path from μ to $m($`big_h`$)$, hence a standard battle from $\iota(\mu)$ to $\iota(m($`big_h`$))$, i.e. from `big_h` to `small_h`.

Since m is assumed to be a variant for standard battles, we get the inequality $m($`small_h`$) < m($`big_h`$)$.

5.4.4 Remarks

We thank J. Ketonen and R. Solovay for the high quality of their explanations and proof details. Our proof follows tightly the sequence of lemmas in their article, with a focus on constructive aspects. Roughly steaking, our implementation *builds*, out of a hypothetic variant m, bounded by some ordinal $\mu < \epsilon_0$, a hydra `big_h` which verifies the impossible inequality $m(\texttt{big\_h}) < m(\texttt{big\_h})$.

On may ask whether the preceding results are not too restrictive, since they refer to a particular data type `T1`. In fact, our representation of ordinals strictly less than ϵ_0 is faithful to their mathematical definition, at least Kurt Schütte's [34], as proved in Chapter 6 on page 163. (please see also le module `Ordinals.Schutte.Injection_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 5.4 with this new definition?

Chapter 6

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.

We chosed as reference an axiomatic definition of the set of countable ordinals, from Kurt Schütte's book " Proof Theory " [34].

Remark 6.1 *In all this chapter, the word "ordinal" will be considered as a synonymous of "countable ordinal"*

Schütte's definition of countable ordinal relies on the following three axioms.

1. There exists a well-ordered set $(\mathbb{O}, <)$
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 of countable ordinals. Unlike the rest of our libraries, the library `Ordinals.Axiomatic` is not constructive, and relies on various axioms.

- First, note 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 withe 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 and Hilbert's ϵ (choice) and Russel's ι (definite description) operators. Both operators allowus 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 sub-library Schutte/ is "contaminated" by various axioms, and the rest of our libraries remains constructive.

6.1 Declarations and Axioms

Let us declare a type `ON` for representing countable ordinals, and a binary relation `lt`. Note that, in our development, `ON` is a type, while the *set* of countable ordinals (called $\mathbb{O}$ by Schütte) is the full set over the type `ON`.

From `Schutte_basics.v`

```
Parameter ON : Type.
Parameter lt : relation ON.
Notation "a < b" := (lt a b): schutte_scope.

Definition ordinal := Full_set ON.
```

Schütte's first axiom tells that `lt` is a well order on the set `ordinal` (The class `WO` is defined in `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 ordinal.
```

The second and third axioms say that a subset X of $\mathbb{O}$ is (strictly) bounded iff it is countable. We use Florian Hatat's library on countable sets, written as he was a student of *École Normale Supérieure de Lyon*.

```
Axiom AX2 : forall X: Ensemble ON,
   (exists a,  (forall y, X y -> y < a)) ->
   countable X.
```

```
Axiom AX3 : forall X : Ensemble ON,
              countable X ->
              exists a,  forall y, In X y -> y < a.
```

AX2 and AX3 could have been replaced by a single axiom (using the iff connector), but we decide to respect asmost as possible the structure of Schütte's definitions.

Besides Schütte's axioms, we needed to admit that the type ON is not empty:

```
Axiom inh_ON : inhabited ON.
```

6.1.1 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.

6.1.1.1 Classical logic

In order to work with classical logic, we import the module Coq.Logic.Classical of Coq's standard library, specifially the following axiom:

```
 Axiom classic : forall P:Prop, P \/ ~P.
```

6.1.1.2 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).

6.1.1.3 The definition of zero

According to the definition of a well order, every non-empty subset of ON has a least element. Furthermore, this least element is unique.

```
Remark R : exists! z : ON, least_member lt  ordinal z.
Proof.
  destruct inh_ON as [a]; apply (well_order (WO:=AX1)) with a .
  split.
Qed.
```

Assume we want to call this element zero.

```
Definition zero : ON.
Proof.
  Fail destruct R.
```

```
The command has indeed failed with message:
Case analysis on sort Type is not allowed for inductive
definition ex.
```

Indeed, the basic logic of Coq does not allow us to eliminate a proof of a proposition $\exists\, x : A,\, P(x)$ for building a term whose type lies in the sort `Type`. The reasons for this impossibility are explained in many documents [5, 17, 21].

Let us import the library `Coq.Logic.Epsilon`, which contains the following axiom.

```
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 `MoreEpsilonIota` and `PartialFun` are extensions of `Coq.logic.Epsilon` for making easier such definitions. See also an article in french [13].

```
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.
```

We are now able de define `zero` as the least ordinal. For this purpose, we define a function returning the least element of any [non-empty] set.

```
Definition the_least {M: Type} {Lt}
           {inh : InH M} {WO: WO Lt} (X: Ensemble M)  : M :=
  the  (least_member  Lt X ).
```

From `Schutte_basics`

```
Definition zero: ON :=the_least ordinal.
```

We want to prove now that zero is less or equal than any ordinal number.

```
Lemma zero_le (alpha : ON) :  zero <= alpha.
Proof.
  unfold zero, the_least, the; apply iota_ind.
```

According to the use of the description operator `iota`, we have to solve two trivial sub-goals.

1. Prove that there exists a unique least member of `ON`
2. Prove that being a least member of `ON` entails the announced inequality

```
2 subgoals (ID 155)

  alpha : ON
  ============================
  exists ! x : ON, least_member lt ordinal x

subgoal 2 (ID 156) is:
 forall a : ON, unique (least_member lt ordinal) a ->
                a <= alpha
```

```
  -  apply the_least_unicity, Inh_ord.
  -  destruct 1 as [[_ H1] _]; apply H1; split.
Qed.
```

6.1.1.4 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 ON).
```

```
 bottom is defined
```

Since we won't be able to prove the proposition `{exists! a: ON, least_member (Empty_set ON) a`, the only properties we would be able to prove about `bottom` would be *trivial* properties, *i.e.* satisfied by *any* element of `ON`, like for instance `bottom = bottom`, or `zero <= bottom`.

```
Lemma le_zero_bottom : zero <= bottom.
Proof. apply zero_le. Qed.

Lemma bottom_eq : bottom = bottom.
Proof. trivial. Qed.

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

```
2 subgoals (ID 413)

  ============================
  exists ! x : ON, least_member lt (Empty_set ON) x

subgoal 2 (ID 414) is:
 forall a : ON, unique (least_member lt (Empty_set ON)) 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 the former more concise and readable. The reader will admit this fact by considering the definitions of `zero`, `succ`, `plus`, etc. in the rest of this chapter.

6.1.2 The successor function

The definition of the function `succ:ON -> ON` is very concise. The successor of any ordinal α is the smallest ordinal strictly greater than α.

```
Definition succ (alpha : ON) := the_least (fun beta => alpha < beta).
```

Using `succ`, we define the folloing predicates.

```
Definition is_succ (alpha:ON) := exists beta, alpha = succ beta.

Definition is_limit (alpha:ON) := alpha <> zero /\ ~ is_succ alpha.
```

It is also easy to define recursively the finite ordinals.

```
Reserved Notation "'F' n" (at level 29) .

Fixpoint finite (i:nat) : ON :=
  match i with
            | 0 => zero
            | S i => succ (F i)
  end
where "'F' i" := (finite i)  : schutte_scope.

Coercion finite : nat >-> ON.
```

How do we prove properties of the successor function? First, we make its specification explicit.

```
Definition succ_spec (alpha:ON) :=
  least_member   lt (fun z => alpha < z).
```

Then, we prove that our function `succ` meets this specification.

```
Lemma succ_ok : forall alpha,  succ_spec alpha  (succ alpha).
Proof.
  intros; unfold succ, the_least, the;  apply iota_spec.
```

```
1 subgoal (ID 172)

  alpha : ON
  ============================
  exists ! x : ON, succ_spec alpha x
```

We have to prove that the set of all ordinals strictly greater than α has a unique least element. But the singleton set $\{\alpha\}$ is countable, thus bounded (by the axiom `AX3`). Hence; the set $\{\beta \in \mathbb{O} | \alpha < \beta\}$ is not empty and therefore has a unique least element.

The Coq proof script is quite short.

```
  destruct (@AX3 (Singleton _ alpha)).
  - apply countable_singleton.
  -  unfold succ_spec; apply the_least_unicity;  exists x; intuition.
Qed.
```

We can"uncap" the description operator for proving properties of the `succ` function.

```
Lemma lt_succ (alpha : ON) :  alpha < succ alpha.
Proof.
  destruct  (succ_ok  alpha);  tauto.
Qed.

Hint Resolve lt_succ : schutte.

Lemma lt_succ_le (alpha beta : ON):
  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 : ON):
  alpha < succ beta -> alpha <= beta.

Lemma succ_mono (alpha beta : ON):
  alpha < beta -> succ alpha < succ beta.

Lemma succ_monoR (alpha beta : ON) :
 succ alpha < succ beta -> alpha < beta.

Lemma lt_succ_lt (alpha beta : ON) :
  is_limit beta ->  alpha < beta -> succ alpha < beta.
```

6.1.3 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 `a` is the least upper bound of `X`.

```
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 ON) : ON  := 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 -> ON`. By `AX3` this range is bounded, hence the set of its upper bounds is not empty and has a least element.

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

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

```
Definition omega := omega_limit finite.
```

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

```
Lemma finite_lt_omega : forall i: nat,  i < omega.

Lemma lt_omega_finite alpha : ON) :
  alpha < omega ->  exists i:nat, alpha =  i.

Lemma is_limit_omega : is_limit omega.
```

6.1.4 Ordering functions

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 or equal than α"

The purpose of this section is to give a meaning to the construction "the α-th element of X" where X is a non empty subset of $\mathbb{O}$. We follow Schütte's approcah, by defining the notion of *ordering functions*, a way to associate a unique ordinal to each element of a given subset of $\mathbb{O}$. Complete definitions and proofs can be found in the module `Ordering_Functions`).

6.1.4.1 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 ON) :=
  forall alpha beta, In A alpha -> beta < alpha -> In A  beta.

Definition proper_segment (A: Ensemble ON) :=
  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 sait to be an *ordering segment* of A. Our definition in Coq is a direct translation of the mathematical text of [34].

```
Definition ordering_function (f : ON -> ON)(A B : Ensemble ON) :=
 segment A /\
 (forall a, In A a -> In B (f a)) /\
 (forall b, B b -> exists a, In A a /\ f a = b) /\
 forall a b, In A a -> In A b -> a < b ->  f a < f b.

Definition ordering_segment (A B : Ensemble ON) :=
  exists f : ON -> ON, 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 ON) :=
  the  (fun x => ordering_segment x B).

Definition ord  (B : Ensemble ON) :=
   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 ordering function to every subset of $\mathbb{O}$.

```
About ordering_function_ex.
```

```
forall B : Ensemble ON,
 exists ! AB : Ensemble ON,
      exists f : ON -> ON, ordering_function f AB B
```

Moreover, the following theorem tells us that two ordering functions of the same set are extentionally equal, which makes our function `ord` non-ambiguous.

```
ordering_function_unicity :
forall (B A1 A2 : Ensemble ON) (f1 f2 : ON -> ON),
ordering_function f1 A1 B ->
ordering_function f2 A2 B -> fun_equiv f1 f2 A1 A2
```

Let us quote the following theorems (see `Ordering_Functions` for more details).

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

Th_13_5_2 :
forall (A B : Ensemble ON) (f : ON -> ON),
ordering_function f A B -> closed B -> continuous f A B
```

6.1.5 Ordinal addition

We are now ready to define and study addition on the type `ON`. The following definitions and proofs can be consulted in `Addition.v`.

```
Definition plus alpha := ord  (ge alpha).
Notation "alpha + beta " := (plus alpha beta) : schutte_scope.
```

In other words, $\alpha + \beta$ is the β-th ordinal greater or equal than α.

Thank 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 : ON) :
  forall P : (ON->ON)->Prop,
    (forall f: ON->ON,
        ordering_function f ordinal (ge alpha)-> P f) ->
    P (plus alpha).
```

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

```
1 subgoal (ID 24)

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

```
(* rest of proof skipped *)
```

The foolowing lemmas are proved the same way.

```
Lemma zero_plus_alpha (alpha : ON) : zero + alpha = alpha.

Lemma le_plus_l (alpha beta : ON) : alpha <= alpha + beta.

Lemma le_plus_r (alpha beta : ON) :  beta <= alpha + beta.

Lemma plus_mono_r (alpha beta gamma : ON) :
    beta < gamma -> alpha + beta < alpha + gamma.

Lemma plus_of_succ (alpha beta : ON) :
    alpha + (succ beta) = succ (alpha + beta).

Theorem plus_assoc (alpha beta gamma : ON) :
  alpha + (beta + gamma) = (alpha + beta) + gamma.

Lemma one_plus_omega :  1 + omega = omega.

Lemma finite_plus_ge_omega (n : nat) (alpha : ON) :
  omega <= alpha -> n + alpha = alpha.
```

It isinteresting to compare the proof of these lemmas with the computational proofs of the corresponding statements in `Epsilon0.T1`. For instance, the proof of the lemma `one_plus_omega` uses the continuity of ordering functions (hence `(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`!

6.1.5.1 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:ON)(n:nat){struct n} :ON :=
 match n with
```

```
            | 0 => alpha
            | S p => mult_Sn alpha p + alpha
 end.

Definition mult_n alpha n :=
  match n with
      0 => zero
    | S p => mult_Sn alpha p
  end.

Notation "alpha * n" := (mult_n alpha n) : schutte_scope.
```

6.1.6 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 $\varphi_0\,\alpha$. 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.

6.1.6.1 Additive principal ordinals

Definition 6.1 *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 `AP.v`

```
Definition AP : Ensemble ON :=
  fun alpha =>
  zero < alpha /\
  (forall beta, beta < alpha ->  beta + alpha = alpha).
```

6.1.6.2 The function `phi0`

Let us call φ_0 the ordering function of `AP`.

```
Definition phi0 := ord AP.
```

In the mathematical text, we shall use indifferently the notations ω^α and$\varphi_0(\alpha)$.

```
  Notation "'omega^'" := phi0 (only parsing) : schutte_scope.
```

6.1.7 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) : ON :=
  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 $\varphi 0$ and the ordinal ϵ_0 (which we could not express within the type `T1`).

6.1.8 Properties of the set `AP`

The set of additive principal ordinals is not empty: it contains at least the ordinals 1 and ω.

```
Lemma AP_one : In AP 1.

Lemma AP_omega : In AP omega.
```

Moreover, 1 is the least principal ordinal and ω is the second element of `AP`.

```
Lemma least_AP: least_member  lt AP 1.

Lemma omega_second_AP :
  least_member   lt
                  (fun alpha => 1 < alpha /\ In AP alpha)
                  omega.
```

The set `AP` is *closed* under addition, and unbounded.

```
 Lemma AP_plus_closed (alpha beta gamma : ON):
     In AP alpha -> beta < alpha -> gamma < alpha -> beta + gamma < alpha.

Theorem AP_unbounded : Unbounded AP.
```

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

```
Theorem AP_closed : closed AP.

Lemma AP_o_segment :  the_ordering_segment AP = ordinal.
```

6.1.8.1 Properties of the function φ_0

The ordering function of the set `AP` is defined on the full set $\mathbb{O}$ and is continuous (Schütte says that this function is *normal*).

```
Theorem normal_phi0 : normal phi0 AP.
```

The following properties come from the definition of φ_0 as the ordering function of `AP`. It may be interesting to compare these proofs with the computational ones described in Chapter 4.

```
Lemma AP_phi0 (alpha : ON) : In AP (phi0 alpha).

Lemma phi0_zero : phi0 zero =  1.

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

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

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

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

Lemma omega_eq : omega = phi0 1.

Lemma phi0_le (alpha : ON) : alpha <= phi0 alpha.
```

Please note that the lemma `omega_eq` above, is consistent with the interpretation of the ordering function φ_0 as the exponential of basis ω. Indeed we could have written this lemma with our alternative notation:

```
Lemma omega_eq : omega = omega^ 1.
```

6.1.8.2 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 the module* ***`Epsilon0.T1`****.*

```
Lemma lt_phi0 (alpha : ON):
  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 or equal than the omega-tower of height j. By monotonicity, $\varphi_0(\alpha)$ is greater or equal than 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.
```

6.1.8.3 Cantor normal form

The notion of Cantor normal form is defined for all countable ordinals. Nevertheless, note that contrary to the implemenation base on type `T1`, the Cantor normal form of an ordinal α may contain α as a sub-term!

Let us comment the main definitions and results of our library `CNF.v`

A Cantor normal form is represented as a list of ordinals.

```
Definition cnf_t := list ON.
```

A given list l is a Cantor normal of a given ordinal α if it satisfies two conditions:

- The list l is sorted (in decreasing order) w.r.t. the order $\leq$
- The sum of all the ω^{β_i} where the β_i are the terms of l (in this order) is equal to α.

```
Fixpoint eval (l : cnf_t) : ON :=
  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 : ON)(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 : ON) :
  exists l: cnf_t, is_cnf_of alpha l.
```

By structural induction on lists, we prove that this normal form is unique.

```
 Lemma cnf_unicity : forall l alpha,
   is_cnf_of alpha l ->
   forall l',  is_cnf_of alpha l' -> l=l'.
Proof.
 induction l.
 (*  end of proof skipped  *)
Qed.

Theorem cnf_exists_unique (alpha:ON) :
  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.

```
Lemma cnf_lt_epsilon0 :
 forall l alpha,
   is_cnf_of alpha l ->
  alpha < epsilon0 ->
  Forall (fun beta =>  beta < alpha) l.
```

```
Lemma cnf_of_epsilon0 : is_cnf_of epsilon0 (epsilon0 :: nil).
Proof.
  split.
  - constructor.
  - simpl;  now rewrite alpha_plus_zero, epsilon0_fxp.
Qed.
```

6.1.9 An embedding of `T1` into `ON`

Our library `Injection_from_T1.v` 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 `ON` by structural recursion.

```
Fixpoint inject (t:T1) : ON :=
 match t with T1.zero => zero
         | T1.ocons a n b =>
           AP.phi0 (inject a) * S n + inject b
 end.
```

This function enjoys good commutation properties with respect to the main operations which allow us to build Cantor normal form.

```
Theorem inject_of_zero : inject T1.zero = zero.

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

Theorem inject_of_phi0 (alpha : T1):
  inject (phi0 alpha) = AP.phi0 (inject alpha).

Theorem inject_plus (alpha beta : T1): nf alpha -> nf beta ->
  inject (alpha + beta)%t1 = inject alpha + inject beta.


Theorem inject_mult_n (alpha : T1)  :
  nf alpha -> forall n:nat , inject (alpha *  n)%t1 =  inject alpha * n.

Theorem inject_mono (beta gamma : T1) :
  T1.lt  beta gamma ->
  T1.nf beta -> T1.nf gamma ->
  inject beta < inject gamma.

Theorem inject_injective (beta gamma : T1) : nf beta -> nf gamma ->
  inject beta = inject gamma -> beta = gamma.
```

Finally, we prove that `inject` is a bijection from the set of all terms of `T1` in normal form to the set `members epsilon0` of the elements of `ON` strictly less than ϵ_0.

```
 Theorem inject_lt_epsilon0 (alpha : T1):
      inject alpha < epsilon0.

Theorem embedding :
     fun_bijection (nf: Ensemble T1)  (members epsilon0) inject.
```

6.1.10 Remarks

Let us recall that this library `Schutte` depends on five *axioms* and lies explicitely 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.

6.1.11 Related work

In [26], José Grimm establishes the consistency between or ordinal notations (`T1` and `T2` (Veblen normal form) and his implementation of ordinal numbers

after Bourbaki's set theory.

Chapter 7

Alpha-large sequences and rapidly growing functions

Todo: Still very incomplete !

7.1 Introduction

Let us assume that Hercules always always chops off the rightmost among the lowest heads. Let $h = \iota(\alpha)$ be a hydra [configuration] where α is some ordinal stricly less than ϵ_0. Then a possible next configuration may be $\iota(\{\alpha\}(i))$, where i is some strictly positive natural number

Then, given some initial configuration, a battle can be determined by a finite sequence of natural numbers $s = s_1, s_2, \ldots, s_N$, where, at the k-th round of the fight, Hercules chops off the rightmost among the lowest heads, and the hydra replies with i_k as the replication number

On the ordinals' side, given an ordinal α and a sequence s, we can compute a sequence of ordinals $\alpha_0 = \alpha \ldots, \alpha_{k+1} = \{\alpha_k\}(s_k) \ldots$. In [27], the last ordinal α_N is denoted by $\{\alpha\}\langle s\rangle$.

Todo: introduce `sorted_ge`: allows us to replace KS's sets by sorted lists and use library `Coq.Sorting.Sorted`

```
(* sorted list of natural numbers greater or equal than n *)

Inductive sorted_ge (n: nat) : list nat -> Prop :=
| sorted_ge_nil : sorted_ge n nil
| sorted_ge_one : forall p, n<=p ->
                            sorted_ge n (p::nil)
| sorted_ge_cons: forall p q s,  n<=p -> p<q ->
                                 sorted_ge p (q::s) ->
                                 sorted_ge n (p::q::s).
```

7.2 Gnawing ordinals

In Coq, we define a function `gnaw` of type `list nat -> T1 -> T1`, such that `gnaw` s α evaluates to $\{\alpha\}\langle s\rangle$.

Todo: justifier le terme "gnaw"

Todo: Cite [27], and comment this adaptation to Coq, lemma by lemma

From teaser.Ordinals.Epsilon0.Alpha_largeS

```
Fixpoint gnaw (s: list nat) (alpha : T1) :=
  match alpha, X with
    | _, nil => alpha
    | _, (0::Y) => gnaw Y alpha
    | _, (S i :: s')  =>  gnaw s' (canonS i alpha)
  end.
```

Remark 7.1 Our definition differs slightly from [27]. Nevertheless, if s is a strictly increasing sequence of strictly positive integers (*i.e.* satisfies our predicate `sorted_ge 1`), then `gnaw` s α returns Ketonen and Solovay's $\{X\}\langle\alpha\rangle$ where X is the range of s.

The following example proves the equality $\{\omega^3 + 2\}\langle 1, 2, \ldots, 303\rangle = \omega^2 + \omega * 93 + 83$

```
Example ex1 :  gnaw (interval 1 303) (omega ^ 3  +  2) =
               omega ^ 2 + omega * 93  + 83.
Proof. reflexivity. Qed.
```

Definition 7.1 *The sequence s is said to be* α-large *if* $\{\alpha\}\langle s\rangle = 0$.

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

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

7.2.1 Some proofs by computation

The function `gnaw` and its derivates `largeb` and `large` are tractable only for small ordinals and sequnces of small integers. Let us *compute* some values.

7.2.1.1 n-large sequences

let us consider a finite non-zero ordinal n (with $0 < n$). For any natural number $i \geq 1$, we have $\{n\}(i) = n - 1$. Thus any sequence of at least n numbers will "gnaw" the ordinal n.

For instance:

```
Compute gnaw  (iota 40) 42.
(* FS 1 *)
```

```
Compute gnaw (iota 41) 42.
(* One *)

Compute gnaw (iota 55) 42.
(* zero *)

Goal  ~large 42 (interval 100 139).
Proof. discriminate. Qed.

Goal  large 42 (interval 100 141).
Proof. reflexivity. Qed.
```

Let us try to compute some values $\{\alpha\}\langle i\rangle$, for small infinite ordinals α.

```
Compute gnaw (interval 1 2) omega.  (* zero *)

Compute gnaw (interval 10 19) omega. (* one *)

Compute gnaw (interval 100 199) omega. (* one *)
```

We can conjecture that, if $0 < i$ then $2 \times i$ is the least natural number j such $\{\omega\}\langle[i..j]\rangle = 0$. Before proving this statement, let us experiment on multiples of ω.

```
Compute gnaw (interval 10 41) (omega + omega). (* 1 *)
Compute gnaw (interval 100 401) (omega + omega). (* 1 *)
Compute gnaw (interval 100 200) (omega + omega). (* omega *)
Compute gnaw (interval 100 805) (omega * 3).  (* one *)
Compute gnaw (interval 20 (8 * 20 + 5)) (omega * 3).  (* 1 *)
Compute gnaw (interval 20 (32 * 20 + 29)) (omega * 5).  (* 1 *)
```

Like any combinatorist, we try to infer a general law from these computations. Let us propose the following one:

Conjecture 7.1 *For any pair (i, j) of strictly positive integers, $2^{j+1} - 2$ is the least k such that the interval $[i..k]$ is $\omega \times i$-large.*

We can still use Coq for *testing* our conjecture.

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

Definition my_test (i j : nat) :=
   gnaw (interval  j (exp2 i * (j + 1) -3)%nat) (omega *  i)
   = 1.
```

```
Compute my_test 4 20.
```

For even small values of α, it seems that testing can be very inefficient for guessing which value of j can make the interval $[i, j]$ α-large.

The following computation shows that $\{\omega^2\}\langle[2, 4444]\rangle = \omega \times 6 + 4258$.

```
Compute gnaw (interval 2 4444) (omega * omega * 2).
(*  = ocons One 5 (FS 4257)
     : T1 *)
```

It seems obvious that gnawing $\omega^2 \times 2$ by the interval $[2, 4258 + 4444]$ will yield $\omega \times 6$.

```
Example Ex :
  gnaw (interval 2 (4444 + 4258)) (omega * omega * 2) = omega * 6.
Proof.  reflexivity. Qed.
```

According to our conjecture, we guess that the value of j we are looking for is $2^6 \times (4444 + 4258 + 2) - 2 = 557054$.

- $\{\omega^2\}\langle 2..4444 + 4258\rangle = \omega \times 6$
- $\{\omega \times 6\}\langle 8703..2^6 \times 8704 - 3\rangle = 1$

TO do : extraction to OCaml with binary integers

But, the numbers are too big for simple computations, and we would replace tests by formal proofs.

Most of the following lemmas come from [27]. Let us start with some properties of $\{i\}\langle k\rangle$ where i a finite ordinal.

7.2.2 n-large sequences

The following lemma generalizes our previous tests.

Lemma 7.1 *For any natural number n and any strictly increasing sequence s of strictly positive integers, s is n-large iff $|s| \geq n$.*

```
Lemma large_n_iff : forall X (n:nat),
  sorted_ge 1 X ->
  large  n X <-> (n <= List.length X)%nat.
```

Todo: Lemmas relating `sorted_ge` and `interval` ?

This lemma corresponds to the first part of proposition 4.2 of [27].

For instance, the interval $[1, i]$ is 42-large if and only if $i \geq 42$.

7.2.2.1 ω-large sequences

Gnawing the ordinal ω with a sequence $n :: s$ reduces to gnaw the finite ordinal n with the sequence s. Thus, we obtain the following lemma

Lemma 7.2 *Any strictly increasing sequence $n :: s$ of strictly positive integers is ω-large iff $|s| \geq n$.*

```
Lemma large_omega_iff : forall s n,  sorted_ge 1 (n:: s) ->
                                     (n <=  List.length s)%nat <->
                                     large omega (n::s).
```

Todo: apply this lemma to an example.

This lemma corresponds to the second part of proposition 4.2 of [27].

```
Example omega_1_2_large : large omega (iota 2).
Proof. reflexivity. Qed.
```

```
Example omega_10_19_not_large : ~ large omega (interval 10 19).
Proof. discriminate. Qed.
```

This is expressed in the following lemma (first part of `Proposition 4.2` of KS [27]).

7.2.3 First Lemmas

`Proposition 4.2` of KS [27] contains the following statement:

> A finite set X is ω-large is and only if $|X| < \min X$

Since we represent sets as strictly increasing sequences of natural numbers, our statement decomposes the "set" X into a non-empty sequence $n :: s$, where n is trivially the least element of X. Thus, the translation into Coq is as follows:

```
Lemma large_omega_iff : forall s n,
    sorted_ge 1 (n::s) ->
    (n <=  List.length s)%nat <->
    large omega (n::s).
```

Chapter 8

Generalities

8.1 Well-foundedness in Standard Library

Fortunately, Coq's standard library gives us a much more direct way to handle termination properties, based on a constructive definition of well-founded relations. Well foundedness is defined in the module `Coq.Init.Wf` of Coq's standard library. It is based on the concept of *accessibility*. Let us quote Coq's source.

```
Section Well_founded.

 Variable A : Type.
 Variable R : A -> A -> Prop.

 Inductive Acc (x: A) : Prop :=
     Acc_intro : (forall y:A, R y x -> Acc y) -> Acc x.
```

According to this definition, an element x of type A is accessible wrt R iff any antecedent of x is accessible. Due to the inductive form of `Acc`'s definition, every proof of accessibility is a proof tree whose branches are finite. Then, we define a relation R to be well-founded if every element $a : A$ is accessible wrt R. The advantages of this approach over the "classical" one are multiple:

- It allows to reason by well-founded induction (also called "transfinite induction"). Technically, well-founded induction comes for free, since it derives from induction over accessiblity proofs
- It is the basis of general recursion in Coq
- It entails the "classical" definition: if R is well-founded, there is no infinite sequence a_i ($i \in \mathbb{N}$) such that $a_{i+1}\, R\, a_i$ for every i.

The interested reader can consult Chap. 15 of [5] and Chap 7 of [17] for more details. It is also worth to replay with Proof General or CoqIde some proofs of `Coq.Wellfounded`.

Show the "rect" induction principle

```
Require Import Wellfounded.

About well_founded_induction.
```

```
well_founded_induction :
forall (A : Type) (R : A -> A -> Prop),
well_founded R ->
forall P : A -> Set,
(forall x : A, (forall y : A, R y x -> P y) -> P x) ->
 forall a : A, P a

Arguments A, R are implicit
Argument scopes are [type_scope _ _ _ _ _]
well_founded_induction is transparent
Expands to: Constant Coq.Init.Wf.well_founded_induction
```

8.1.1 A simple direct proof of well-foundedness

The best way to feel how well-foundedness works in Coq is to look at a direct proof of `Arith.Wf_nat.lt_wf`, which states that the strict order $<$ over the set of natural numbers is well founded.

Our strategy consists in proving by induction on `n` that any element `n` is accessible wrt `lt`.

```
Require Import Lt.

Theorem lt_wf : well_founded lt.
Proof.
   intro n; induction n as [ | p IHp].
```

The first subgoal consists in proving that 0 is accessible, i.e. that any element `y` such that `y < 0` is accessible. The tactics `inversion_clear`, allied to the hypothesis `y < 0` solves this subgoal immediately.

```
 - split; inversion_clear  1.
```

The inductive step of the proof is as follows:

```
  p : nat
  IHp : Acc lt p
  ============================
   Acc lt (S p)
```

For proving that `S p` is accessible, it sufices to prove that any antecedent of `S p` is accessible.

```
- split; intros y Hy.
```

```
  p : nat
  IHp : Acc lt p
  y : nat
  Hy : y < S p
  ============================
   Acc lt y
```

But the analysis of `Hy` gives us two cases : either `y = p`, and `y` is accessible by `IHp`, or `S y <= p`, in which case `y < p`, which implies `Acc lt y` by definition.

```
  inversion_clear Hy .
    + assumption.
    + assert (y < p) by  auto with arith.
      destruct IHp; auto.
Qed.
```

8.1.2 Using operators on relations

Direct proofs of accessibility, and thus of well-foundedness can be more complex than the previous example. One of the lemmas that entail the termination of any hydra battle is a quite complex proof of accessibility (in Sect 4.3.1.2 on page 134). Happily, library `Coq.Wellfounded` provides us with a set of tools, adapted from L.Paulson [30], that make well-foundedness proofs easy if we can decompose the considered relation according to some operators. In other words, the difficult parts of the proof of accessibility are encapsulated in *generic* lemmas one has "just" to apply.

Let us look at some examples.

8.1.3 Relation inclusion

First, let us prove that the relation defined by $n\,R\,p$ iff $0 < n \wedge p = 2n$ is well founded. For that purpose, we apply the theorem `wf_incl` of library `Coq.Wellfounded.Inclusion`, which states that if a relation R is well-founded, then any relation S such that $S \subseteq R$ is well-founded too.

```
Definition R (n p :nat) := 0 < n /\ p = 2 * n.

Require Import Wellfounded.Inclusion Omega.

Lemma Rwf : well_founded R.
Proof.
  apply wf_incl with lt.
```

Two sub-goals remain to be solved. The first one, is proving that our relation `R` is included in the strict order `lt` on type `nat`.

```
- intros n p  [H H0].
```

The tactic `omega` solves this subgoal immediately.

The second sub-goal corresponds to proving that `lt` is well-founded. But this is already proven.

```
1 focused subgoal
  ============================
   well_founded lt
```

```
  - apply lt_wf.
 Qed.
```

Exercice 8.1 Try to write a direct, but simple proof of `Rwf` (i.e. without using `wf_incl`). If this task becomes too complex, you may abort your attempt, and consider the use of `wf_incl` as the best way to prove this theorem.

8.1.4 Inverse image

Let us consider another example: proving that the relation "being a strict prefix" on finite lists, is well-founded. First, we define formally this relation:

```
Require Import List.
Open Scope list_scope.


Inductive strict_prefix (A:Type)  : relation (list A) :=
strict_prefix_intro :
    forall  (a:A) l l',  strict_prefix _  l (l ++ (a::l')).
Arguments strict_prefix_intro {A} _ _ _.
```

Our strategy is to reduce this property to a comparison of list lenghts. If l is a strict prefix of l', then l must be shorter than l'. We start the proof with an application of `wf_incl`.

```
Theorem strict_prefix_wf {A:Type} : well_founded (strict_prefix A).
Proof.
  apply wf_incl with (fun l l' =>  length l < length l').
  - intros l l' H.  destruct H as [a].
    SearchRewrite (length (_ ++ _)).
```

```
app_length:
  forall (A : Type) (l l' : list A),
   length (l ++ l') = length l + length l'
```

```
   rewrite app_length;cbn; omega.
```

```
1 subgoal, subgoal 1 (ID 24)

subgoal 1 (ID 24) is:
 well_founded (fun l l' : list A => length l < length l')
```

The theorem `wf_inverse_image` of library `Coq.WellFounded.Inverse_Image` allows us to apply `lt_wf`.

```
wf_inverse_image :
forall (A B : Type) (R : B -> B -> Prop) (f : A -> B),
well_founded R -> well_founded (fun x y : A => R (f x) (f y))
```

```
  - apply wf_inverse_image, lt_wf.
Qed.
```

```
  strict_prefix_wf is defined.
```

Remark 8.1 The last proof uses two theorems from `Coq.Wellfounded`: `wf_inverse_image` and `wf_incl`. This combination is a frequently used pattern. We will often call *measure* a function `m` such that whenever `R x y`, `S (m x) (m y)` holds. If `S` is well-founded, then `R` is well-founded too.

8.1.5 Lexicographic product

Library `Coq.WellFounded.Lexicographic_Product` contains a definition of *dependent* lexicographic product, as well as a proof that well-founded relations are closed under this operation. For simplicity's sake we derived from that module a non-dependent version in module `Prelude.Simple_LexProd`.

```
(** Non dependent lexicographic product *)

Section Definitions.

  Variables (A B : Type)
            (ltA : relation A)
            (ltB : relation B).

  Hypothesis wfA : well_founded ltA.
  Hypothesis wfB : well_founded ltB.

  Inductive lexico : relation (A * B) :=
   lex_1 : forall a a' b b', ltA a a' -> lexico (a,b) (a',b')
|  lex_2 : forall a  b b', ltB b b' -> lexico (a,b) (a,b') .
```

```
(** Non dependent lexicographic product *)

Section Definitions.

  Variables (A B : Type)
            (ltA : relation A)
            (ltB : relation B).

  Hypothesis wfA : well_founded ltA.
  Hypothesis wfB : well_founded ltB.

  Inductive lexico : relation (A * B) :=
   lex_1 : forall a a' b b', ltA a a' -> lexico (a,b) (a',b')
|  lex_2 : forall a  b b', ltB b b' -> lexico (a,b) (a,b') .
 Lemma lexico_wf : well_founded lexico.
(* proof by reduction to dependent lexicographic product (omitted) *)
```

It is now trivial to prove for instance that the lexicographic product on `nat * nat * nat` is well-founded.

```
Require Import Simple_LexProd.

Theorem lt2_wf : well_founded (lexico lt (lexico lt lt)).
Proof.
 repeat apply lexico_wf; apply  lt_wf.
Qed.
```

Exercice 8.2 Consider the following relation on $\mathbb{N}\times\mathbb{N}$, defined by the following propositions (for any n and p).

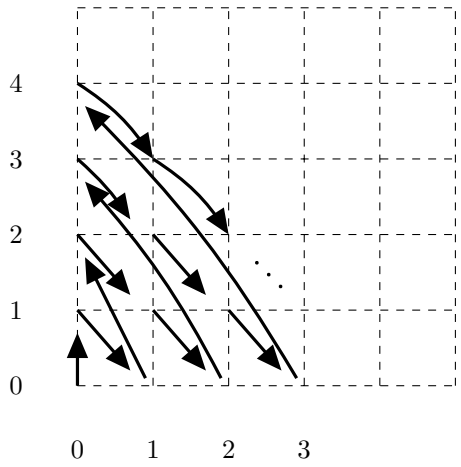
$$(n, S\ p) \longrightarrow (S\ n, p) \tag{8.1}$$
$$(n, 0) \longrightarrow (0, S\ n) \tag{8.2}$$

Fig. 8.1 on the next page represents a small part of this relation. It is closely related to Cantor's enumeration of $\mathbb{N} \times \mathbb{N}$: two pairs (n, p) and (q, r) are related through $\longrightarrow$ if (q, r) is the successor of (n, p) in that enumeration. This makes well-foundedness of $\longrightarrow$ *intuitively* obvious. Nevertheless we want to write formal proofs of this property.

1. Give an inductive definition of $\longrightarrow$
2. Give several proofs of its well foundedness:
   - A direct proof, by nested induction on **nat** The main induction could be on the sum $n + p$, and the inner induction on the number n
   - A proof using a measure m from **nat*nat** into **nat**: Prove that m cannot be a linear function, i.e. of the form `fun p:nat * nat => a * fst p + b * snd p`

- A proof using a measure from **nat*nat** into **nat*nat**, equipped with the standard lexicographic ordering. **Hint**: The measure m may be the inverse function of the famous bijection from $\mathbb{N} \times \mathbb{N}$ into $\mathbb{N}$.

Figure 8.1: First steps of the enumeration of $\mathbb{N} \times \mathbb{N}$



Chapter 9

Appendices

9.1 How to install the libraries

- The present distribution has been checked with version 8.9.1 of the Coq proof assistant
- just go into the `Teaser` directory, and type "`make`"

9.2 Contents of the main files

9.2.1 Exponentiation algorithms

9.2.1.1 Powers.Pow

Module teaser.Powers.Pow defines two polymorphic functions for computing x^n: the naïve (*i.e.* linear) one and the binary method, that takes less than $2 \times \log_2(n)$ multiplications.

9.2.2 Hydras and Ordinal Numbers

9.2.2.1 Directory Ordinals/Epsilon0

Epsilon0.v Data structure for Cantor normal form (from the Castéran-Contejean contribution)

Epsilon0rpo.v Proof of wellfoundness of the order on Cantor normal form (from the "Cantor" contribution [14])

NaturalSum.v The natural (commutative) addition on ordinals less than ϵ_0 (a.k.a Hessenberg's sum).

KS.v On canonical sequences of ordinals (the *Ketonen-Solovay machinery*)

Alpha_largeS.v On α-large sequences (*still very unstable development*).

9.2.2.2 Directory Ordinals/Gamma0

This directory contains some basic definitions on Veblen normal forms.

Gamma0 Veblen normal forms (to be completed).

9.2.2.3 Directory Ordinals/Hydra

- Definition of hydra battles.
- Proof of termination of all hydra battles.
- Proof that the ϵ_0 is the least ordinal that can be used for proving the termination of all hydra battles (considering variants).

9.2.2.4 Directory Ordinals/Prelude

A lot of auxiliary definitions.

9.2.2.5 Directory Ordinals/rpo

Properties of the recursive path ordering (contributed by Évelyne Contejean).

9.2.2.6 Directory Ordinals/Axiomatic

This directory contains the axiomatisation of countable ordinals, after K. Schütte.
It is written in classical logic, using Hilbert's ϵ operator.

Schutte.v The Axioms by Schütte; first results

Ordering_Functions.v Ordering functions

Plus.v Ordinal addition

AP.v Additive principal ordinals

CNF.v Existence and unicity of Cantor normal form

Critical.v Critical ordinals

Injection_From_T1.v Correspondance between ordinal notations and Schütte's ordinal (up to ϵ_0)

9.2.2.7 Directory Ordinals/Drafts

Experimental stuff. To develop!

Bibliography

[1] J.-R. Abrial. *The B-book: Assigning Programs to Meanings*. Cambridge University Press, New York, NY, USA, 1996.

[2] Jean-Raymond Abrial. *Modeling in Event-B: System and Software Engineering*. Cambridge University Press, New York, NY, USA, 1st edition, 2010.

[3] Andrej Bauer. The hydra game. https://github.com/andrejbauer/hydra, 2008.

[4] Jean Berstel and Srecko Brlek. On the length of word chains. *Inf. Process. Lett.*, 26(1):23–28, 1987.

[5] Yves Bertot and Pierre Castéran. *Interactive Theorem Proving and Program Development. Coq'Art: The Calculus of Inductive Constructions*. Springer, 2004. `http://www.labri.fr/perso/casteran/CoqArt/index.html`.

[6] Alfred Brauer. On addition chains. *Bull. Amer. Math. Soc.*, 45(10):736–739, 10 1939.

[7] Srecko Brlek, Pierre Castéran, Laurent Habsieger, and Richard Mallette. On-line evaluation of powers using euclid's algorithm. *ITA*, 29(5):431–450, 1995.

[8] Srecko Brlek, Pierre Castéran, and Robert Strandh. On addition schemes. In *TAPSOFT'91: Proceedings of the International Joint Conference on Theory and Practice of Software Development, Brighton, UK, April 8-12, 1991, Volume 2: Advances in Distributed Computing (ADC) and Colloquium on Combining Paradigms for Software Development (CCPSD)*, pages 379–393, 1991.

[9] Daniel Brown. Parametricity. http://www.ccs.neu.edu/home/matthias/369-s10/Transcript/parametricity.pdf, 2010. Available on Matthias Felleisen page.

[10] William H. Burge. *Recursive programming techniques / William H. Burge*. Addison-Wesley Pub. Co Reading, Mass, 1975.

[11] Georg Cantor. *Contributions to the Founding of the Theory of Transfinite Numbers*. Courier Corporation, 1955.

[12] Pierre Castéran. Additions. User Contributions to the Coq Proof Assistant.

[13] Pierre Castéran. Utilisation en Coq de l'opérateur de description. In *Actes des Journées Francophones des Langages Applicatifs*, 2007. http://jfla.inria.fr/2007/actes/index.html.

[14] Pierre Castéran and Évelyne Contéjean. On ordinal notations. User Contributions to the Coq Proof Assistant, 2006.

[15] Pierre Castéran and Matthieu Sozeau. A gentle Introduction to Type Classes and Relations in Coq. http://www.labri.fr/perso/casteran/CoqArt/TypeClassesTut/typeclassestut.pdf.

[16] Adam Chlipala. Parametric higher-order abstract syntax for mechanized semantics. In *Proceedings of the 13th ACM SIGPLAN International Conference on Functional Programming (ICFP'08)*, 2008.

[17] Adam Chlipala. *Certified Programming with Dependent Types*. MIT Press, 2011. `http://adam.chlipala.net/cpdt/`.

[18] Cyril Cohen, Maxime Dénès, and Anders Mörtberg. Refinements for Free! In *Certified Programs and Proofs*, pages 147 – 162, Melbourne, Australia, December 2013.

[19] Cyril Cohen, Maxime Dénès, and Anders Mörtberg. Refinements for free! In Georges Gonthier and Michael Norrish, editors, *Certified Programs and Proofs - Third International Conference, CPP 2013, Melbourne, VIC, Australia, December 11-13, 2013, Proceedings*. Springer, Springer, 2013.

[20] Hubert Comon-Lundh, Claude Kirchner, and Hélène Kirchner, editors. *Rewriting, computation and proof : essays dedicated to Jean-Pierre J ouannaud on the occasion of his 60th birthday*. Lecture Notes in Computer Science. Springer, Berlin, New York, 2007.

[21] Coq Development Team. The coq proof assistant. coq.inria.fr.

[22] Nachum Dershowitz and Georg Moser. The hydra battle revisited. In Hubert Comon-Lundh, Claude Kirchner, and Hélène Kirchner, editors, *Rewriting, Computation and Proof: Essays Dedicated to Jean-Pierre Jouannaud on the Occasion of His 60th Birthday*, pages 1–27. Springer Berlin Heidelberg, Berlin, Heidelberg, 2007.

[23] Thomas Hales et al. A formal proof of the Kepler conjecture. https://arxiv.org/abs/1501.02155, 2015.

[24] Georges Gonthier. Formal proof — the four-color theorem. *Notices of the American Mathematical Society*, 55(11), December 2008.

[25] R. L. Goodstein. On the restricted ordinal theorem. *Journal of Symbolic Logic*, 9(2):33–41, 1944.

[26] José Grimm. Implementation of three types of ordinals in Coq. Research Report RR-8407, INRIA, 2013.

[27] Jussi Ketonen and Robert Solovay. Rapidly growing Ramsey functions. *Annals of Mathematics*, 113(2):267–314, 1981.

[28] Laurie Kirby and Jeff Paris. Accessible independence results for Peano arithmetic. *Bulletin of the London Mathematical Society*, 14:725–731, 1982.

[29] Panagiotis Manolios and Daron Vroon. Ordinal arithmetic: Algorithms and mechanization. *Journal of Automated Reasoning*, 34(4):387–423, May 2005.

[30] Lawrence C Paulson. Constructing recursion operators in intuitionistic type theory. *Journal of Symbolic Computation*, 2:325–355, 1984.

[31] Benjamin Pierce et al. Software foundations. https://softwarefoundations.cis.upenn.edu/.

[32] John C. Reynolds. Types, abstraction and parametric polymorphism. In R. E. A. Mason, editor, *Information Processing 83*, pages 513–523, Amsterdam, 1983. Elsevier Science Publishers B. V. (North-Holland).

[33] John C. Reynolds. The discovery of continuations. *Lisp and Symbolic Computation*, 6:233–247, 1993.

[34] Kurt Schutte. *Proof theory / Translation from the German by J. N. Crossley.* Springer-Verlag Berlin ; New York, 1977.

[35] Will Sladek. The Termite and the Tower: Goodstein sequences and provability in pa. `www.uio.no/studier/emner/matnat/ifi/INF5170/v08/undervisningsmateriale/sladekgoodstein.pdf`, 2007.

[36] M. Sozeau and N. Oury. First-class type classes. In *TPHOLs '08: Proceedings of the 21st International Conference on Theorem Proving in Higher Order Logics*, pages 278–293, Berlin, Heidelberg, 2008. Springer-Verlag.

[37] Bas Spitters and Eelis van der Weegen. Type classes for mathematics in type theory. http://arxiv.org/pdf/1102.1323.pdf, 2011.

[38] Christopher Strachey. Fundamental concepts in programming languages. *Higher Order Symbol. Comput.*, 13(1-2):11–49, April 2000.

[39] Gerard Tel. *Introduction to Distributed Algorithms.* Cambridge University Press, 2 edition, 2000.

[40] Philip Wadler. Theorems for free! In *Proceedings of the Fourth International Conference on Functional Programming Languages and Computer Architecture*, FPCA '89, pages 347–359, New York, NY, USA, 1989. ACM.

Todo: The index should be an important part of the document!