Addition chains

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

September 5, 2020

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 [Gon08] and the Kepler conjecture [H$^+$15].

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.
- 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 [Coq]. First chapters of textbooks like *Interactive Theorem Proving and Program Development* [BC04], *Software Foundations* [P+] or *Certified Programming with Dependent Types* [Chl11] will give you the right background.

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

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

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

1.0.0.2 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.
```

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 exercise allows us to address advanced programming techniques such as:

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

2.2 Some basic implementations

Let us start with a very naive way of computing the n-th power of x, where n is a natural number and x belongs to some type for which a multiplication and an identity element are defined.

From Module ***Powers.FirstSteps***

```
Section Definitions.

Variables (A: Type)
          (mult: A -> A -> A)
          (one: A).
Local Infix "*" := mult.
Local Notation "1" := one.
```

```
Fixpoint power (x:A)(n:nat) : A :=
  match n with 0%nat => 1
             | S p =>   x * x ^ p
  end
```

```
where "x ^ n" := (power x n).
```

```
Compute power Z.mul 1%Z 2%Z 10.
```

```
   = 1024%Z
     : Z
```

```
Open Scope string_scope.
Compute power append  "" "ab"  12.
```

```
 = "abababababababababababab"
     : string
```

The number of multiplications needed to compute x^n via this function is linear with respect to 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-naive 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 role 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 9, we can implement this algorithm in Gallina. The following definitions are still within the scope of the section open in 2.2 on page 9.

From Module Powers.FirstSteps

```
Fixpoint binary_power_mult (x a:A)(p:positive) : A
  :=
  match p with
    | xH =>  a * x
    | xO q => binary_power_mult  (x * x) a q
    | xI q =>  binary_power_mult  (x * x) (a * x) q
  end.

Fixpoint Pos_bpow (x:A)(p:positive) :=
 match p with
  | xH => x
  | xO q => Pos_bpow  (x * x) q
  | xI q => binary_power_mult   (x * x) x q
end.
```

```
Definition N_bpow x (n:N) :=
  match n with
  | 0%N => 1
  | Npos p => Pos_bpow x p
  end.

End Definitions.
```

Let us close the section `Definitions` and mark the argument `A` as implicit.

```
End Definitions.

Arguments N_bpow {A}.
Arguments power {A}.
```

Remark 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 15).

2.2.3 Examples of computation

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

```
Compute N_bpow Z.mul 1%Z 2%Z 10.
```

```
 = 1024%Z
     : Z
```

```
Require Import String.
Open Scope string_scope.

Compute N_bpow append "" "ab"  12.
```

```
  = "abababababababababababab"
     : string
```

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 [Coq].

A 2×2 matrix will be simply represented by a structure with four fields; each field `c`ij is associated with the i-th line and j-th column of the considered matrix.

```
Module M2.
Section Definitions.

  Variables (A: Type) (zero one : A)  (plus mult  : A -> A -> A).

  Variable rt : semi_ring_theory  zero one plus mult   (@eq A).
```

```
  Add Ring Aring : rt.

  Notation "0" := zero.
  Notation "1" := one.
  Notation "x + y" := (plus x y).
  Notation "x * y " := (mult x y).

  Structure t : Type := mat{c00 : A;  c01 : A;  c10 : A;  c11 : A}.
```

The structure type `M2.t` allows us to define the product of two matrices.

```
Definition M2_mult (M M':t) : t := mat
  (c00 M * c00 M' + c01 M * c10 M') (c00 M * c01 M' + c01 M * c11 M')
  (c10 M * c00 M' + c11 M * c10 M') (c10 M * c01 M' + c11 M * c11 M').
```

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

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

End M2_Definitions.
End M2.
```

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 10).

```
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 "naive" function being a *specification* of any polymorphic exponentiation function.

First, we define a type for any such function.

```
Definition power_t := forall (A:Type)
                             (mult : A -> A -> A)
                             (one:A)
                             (x:A)
                             (n:N), A.
```

Then, we would say that a function `f:power_t` is a correct exponentiation function if it is extensionally equal to `power`.

```
Module Bad.

 Definition correct_expt_function (f : power_t) : Prop :=
  forall A (mult : A -> A -> A) (one:A)
           (x:A) (n:N),
           power mult one x (N.to_nat n) = f A mult one x n.
```

Unfortunately, our definition of `correct_expt` is too general. It suffices to build an interpretation where the multiplication is not associative or `one` is not a neutral element to obtain different results through the two functions.

```
Section CounterExample.
    Let mul (n p : nat) := n + 2 * p.
    Let one := 0.

    Remark mul_not_associative :
      exists  n p q,  mul n (mul p q) <> mul (mul n p) q.
    Proof.
        exists 1, 1, 1; discriminate.
    Qed.

    Remark one_not_neutral  :
      exists n : nat, mul one n <> n.
    Proof.
      exists 1; discriminate.
    Qed.

    Lemma correct_expt_too_strong :
          ~ correct_expt_function (@N_bpow).
    Proof.
```

```
      intro H; specialize (H _ mul one 1  7%N).
      discriminate H.
    Qed.

End CounterExample.
End Bad.
```

So, we will have to improve our definition of correctness, by restricting the universal quantification to associative operations and neutral elements, *i.e.*, by considering *monoids*. An exponentiation function will be considered as correct if it returns always the same result as `power` *in any monoid*.

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 *type classes*[SO08, SvdW11]. The tutorial on type classes [CS] gives more details on type classes and operational type classes, also illustrated with the monoid structure.

First, we define a class and a notation for representing multiplication operators, then we use these definitions for defining the `Monoid` type class.

2.3.1 A common notation for multiplication

Operational type classes [SvdW11] allow us to define a common notation for multiplication in any algebraic structure. First, we associate a class to the notion of *multiplication* on any type A.

From Module Powers/Monoid_def.v.

```
Class Mult_op (A:Type) := mult_op : A -> A -> A.
```

From the type theoretic point of view, the term (`Mult_op` A) is $\beta\delta$-reducible to $A \to A \to A$, and if *op* has type (`Mult_op` A), then (`@mult_op A` *op*) is convertible with *op*.

We are now ready to define a new notation scope, in which the notation `x * y` will be interpreted as an application of the function `mult_op`.

```
Delimit Scope M_scope with M.
Infix "*" := mult_op : M_scope.
Open Scope M_scope.
```

Let us show two examples of use of the notation scope `M_scope`. Each example consists in declaring an instance of `Mult_op`, then type checking or evaluating a term of the form `x * y` in `M_scope`.

Note that, since the reserved notation `"_ * _ "` is present in several scopes such as `nat_scope`, `Z_scope`, `N_scope`, etc., in addition to `M_scope`, the user should take care of which scopes are active — and with which precedence — in a Gallina term. In case of doubt, explicit scope delimiters should be used.

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

From Module Powers.Monoid_def

```
Set Printing All.
Check  3 * 4.
```

```
@mult_op nat nat_mult_op (S (S (S O))) (S (S (S (S O))))
     : nat
```

```
Unset Printing All.
Compute 3 * 4.
```

```
 = 12  : nat
```

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 `@mult_op string ?X "abc" "def"`, then the type class mechanism replaces the unknown ?X with `string_op`.

From Module Powers.Monoid_def

```
Require Import String.

Instance string_op : Mult_op string := append.
Open Scope string_scope.

Example ex_string : "ab" * "cde" = "abcde".
Proof. reflexivity. Qed.
```

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 explicitly providing some arguments of `mult_op`. For instance, in Sect. 2.3.3.2 on the following page, we consider various monoids on types `nat` and `N`.

2.3.2 The Monoid Type Class

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

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

```
Class Monoid {A:Type}(op : Mult_op A)(one : A) : Prop :=
{
    op_assoc : forall x y z:A, x * (y * z) = x * y * z;
    one_left : forall x, one * x = x;
    one_right : forall x, x * one = x
}.
```

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 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 39 for more details.

```
Instance nat_plus_op : Mult_op nat | 12 := Nat.add.

Instance Natplus : Monoid nat_plus_op  0%nat | 12.
(* Proof omitted *)
```

Similarly, instances `NMult` and `NPlus` are built for type `N`, and `PMult` for type `positive`.

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 12, 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 | n < m}`, it looks simpler to compute with numbers of type `N` and consider the multiplication $x \times y \mod m$.

It is easy to prove that this operation is associative, using library `NArith`. Unfortunately, the following proposition is false in general (left as an exercise).

$$\forall x : N, (1 * x) \mod m = x$$

Thus, we define a more general class, parameterized by an equivalence relation `Aeq` on a type `A`, compatible with the multiplication `*`. The laws of associativity and neutral element are not expressed as Leibniz equalities but as equivalence statements:

First, let us define an operational type class for equivalence relations:

From Module Powers.Monoid_def

```
Class Equiv A := equiv : relation A.

Infix "==" := equiv (at level 70) : type_scope.
```

The definition of class `EMonoid` looks like `Monoid`'s definition, plus some constraints on `E_eq`.

Please look for instance at our tutorial on type classes and relations [CS] for understanding the use of type classes `Equivalence`, `Reflexive`, `Proper`, etc, in relation with tactics like `rewrite`, `reflexivity`, etc., in proofs which involve equivalence relations instead of equality.

```
Class EMonoid (A:Type)(E_op : Mult_op A)(E_one : A)
      (E_eq: Equiv A): Prop :=
  {
    Eq_equiv :> Equivalence equiv;
    Eop_proper :> Proper (equiv ==> equiv ==> equiv) E_op;
    Eop_assoc : forall x y z, x * (y * z) == x * y * z;
    Eone_left : forall x,  E_one * x == x;
    Eone_right : forall x,  x * E_one ==  x
  }.
```

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.
```

Remark 2.1 In the definition of `Monoid_EMonoid`, the free variables `A`, `op` and `one` are automatically generalized thanks to the *backquote* syntax (see the section about implicit generalization in the reference manual [Coq]).

Thanks to the following *coercion*, every instance of `Monoid` can now be considered as an instance of `EMonoid`. For more details, please look at the section *Implicit Coercions* of Coq's reference manual [Coq].

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

From Module Powers.Monoid_instances

```
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 than or equal to 2. For simplicity's sake, we represent such values using the `N` type, and consider "equivalence modulo m" instead of equality. Note that the law of associativity has been stated as Leibniz' equality.

```
Section Nmodulo.
  Variable m : N.
  Hypothesis m_gt_1 : 1 < m.

  Definition mult_mod ( x y : N) := (x * y) mod m.
  Definition mod_eq ( x y: N) := x mod m = y mod m.

  Global Instance mod_equiv : Equiv N := mod_eq.

  Global Instance mod_op : Mult_op N := mult_mod.

  Global Instance mod_Equiv : Equivalence mod_equiv.
  (* Proof omitted *)
```

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

  Local Open Scope M_Scope.

  Lemma mult_mod_associative :
  forall x y z,  x * (y * z) = x * y * z.
  (* Proof omitted *)

  Lemma one_mod_neutral_l  : forall x, 1 * x ==  x.
  (* Proof omitted *)

  Lemma one_mod_neutral_r  : forall x, x * 1 == x.
  (* Proof omitted *)

  Global Instance Nmod_Monoid : EMonoid  mod_op 1 mod_equiv.
  (* Proof omitted *)

End Nmodulo.
```

2.3.5.2.1 Example In the following interaction, we show how to instantiate the parameter m to a concrete value, for instance 256.

```
Section S256.
Let mod256 :=  mod_op 256.
Local Existing Instance mod256 | 1.

Compute (211 * 67)
```

```
= 57 : N
```

```
End S256.
```

Outside the section `S256`, the term `(211 * 67)%M` is interpreted as a plain multiplication in type `N`:

```
Compute (211 * 67)%M.
```

```
= 14137 : N
```

2.4 Computing Powers in any EMonoid

The module Powers.Pow defines two functions for exponentiation on any `EMonoid` on carrier A. They are essentially the same as in Sect. 2.2 on page 9. 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 naive (linear) Algorithm

The new version of the linear exponentiation function is as follows:

```
Fixpoint power`{M: @EMonoid A  E_op E_one E_eq}
               (x:A) (n:nat) :=
match n with
| 0%nat => E_one
| S p =>   x * x ^ p
end
where "x ^ n" := (power x n) : M_scope.
```

The three following lemmas will be used by the `rewrite` tactic in further correctness proofs. Note that the first two lemmas are strong (*i.e.*, Leibniz) equalities, whilst `power_eq3` is only an equivalence statement, because its proof uses one of the `EMonoid` laws, namely `Eone_right`.

```
Lemma power_eq1 {A:Type} `{M: @EMonoid A  E_op E_one E_eq}
               (x:A) :  x ^ 0 = E_one.
Proof. reflexivity. Qed.

Lemma power_eq2 {A:Type} `{M: @EMonoid A  E_op E_one E_eq}
                (x:A) (n:nat) :
                x ^ (S n)  = x * x ^ n.
Proof. reflexivity. Qed.

Lemma power_eq3 {A:Type} `{M: @EMonoid A  E_op E_one E_eq}
                (x:A) : x ^ 1 == x.
Proof. cbn; rewrite Eone_right; reflexivity. Qed.
```

2.4.1.1 Examples of computation

In the following computations, we first show an exponentiation in $\mathbb{Z}$, then in the type of 31-bit machine integers.[1]

From Module Powers.Demo_power

```
Open Scope M_scope.

Compute 22%Z ^ 20.
```

```
= 705429498686404044207947776%Z
```

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

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

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

[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`.

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 11).

From Module Powers.Pow

```
Fixpoint binary_power_mult `{M: @EMonoid A E_op E_one E_eq}
             (x a:A)(p:positive) : A
  :=
  match p with
    | xH =>    a * x
    | xO q => binary_power_mult  (x * x) a q
    | xI q =>   binary_power_mult (x * x) (a * x) q
  end.

Fixpoint Pos_bpow  `{M: @EMonoid A E_op E_one E_eq}
         (x:A)(p:positive) :=
 match p with
  | xH => x
  | xO q => Pos_bpow  (x * x) q
  | xI q => binary_power_mult (x * x) x q
end.
```

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

From Module Powers.Pow

```
Definition N_bpow {A} `{M: @EMonoid A E_op E_one E_eq} x (n:N) :=
  match n with
  | 0%N => E_one
  | Npos p => Pos_bpow x p
  end.

Infix "^b" := N_bpow (at level 30, right associativity): M_scope.
```

2.4.3 Refinement and Correctness

We have got two functions for computing powers in any monoid. So, it is interesting to ask oneself whether this duplication is useful, and which would be the respective role of `N_bpow` and `power`.

- The function `power`, although very inefficient, is a direct translation of the mathematical definition, as shown by lemmas `power_eq1` to `power_eq3`. Moreover, its structural recursion over type `nat` allows simple proofs by induction over the exponent. Thus, we will consider `power` as a *specification* of any exponentiation algorithm.
- Functions `N_bpow` and `Pos_bpow` are more efficient, but less readable than `power`, and we cannot use these functions before having proved their correctness. In fact, the correctness of `N_bpow` and `Pos_bpow` will mean "being extensionally equivalent to `power`". For instance `N_bpow`'s correctness is expressed by the following statement (in the context of an `EMonoid` on type `A`).

From Module Powers.Pow

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

The relationship between `power` and `N_bpow` can be considered as a kind of *refinement* as in the B-method [Abr96]. Note that the two representations of natural numbers and the function `N.to_nat` form a kind of *data refinement* [Abr10, CDM13a].

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

Section `M_given` of Module Powers.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  ||
    rewrite Eone_right  ||
    rewrite Eop_assoc .

Ltac monoid_simpl := repeat monoid_rw.

Section About_power.
```

In order to make possible proof by rewriting on expressions which contain the exponentiation operator, we have to prove that, whenever x `==` y, the equality x^n `==` y^n holds for any exponent n. For this purpose, we use the `Proper` class of module Coq.Classes.Morphisms

```
Global Instance power_proper :
     Proper (equiv ==> eq ==> equiv) power.
(* Proof omitted *)
```

In the following proofs, we note how notations, type classes and generalized rewriting can be used to write algebraic properties in a nice way.

```
Lemma power_of_plus :   forall x n p, x ^ (n + p) ==  x ^ n *  x ^ p.
(* Proof omitted *)

Ltac power_simpl :=
    repeat (monoid_rw || rewrite <- power_x_plus).
```

Please note that the following two lemmas *do not require* the operation * to be commutative.

```
Lemma power_commute :
    forall x n p, x ^ n * x ^ p ==  x ^ p * x ^ n.
(* Proof omitted *)

Lemma power_commute_with_x :
    forall x n,  x * x ^ n == x ^ n * x.
(* Proof omitted *)

Lemma power_of_power :
   forall x n p,  (x ^ n) ^ p == x ^ (p * n).
(* Proof omitted *)
```

The following two equalities are auxiliary lemmas for proving correctness of the binary exponentiation functions.

```
Lemma sqr_def : forall x, x ^ 2 ==  x * x.
(* Proof omitted *)

Lemma power_of_square :
  forall x n, (x * x) ^ n ==  x ^ n * x ^ n.
(* Proof omitted *)
```

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.

From Module Powers.Pow

```
Lemma binary_power_mult_ok :
  forall p a x,   binary_power_mult  x a p  ==
                  a * x ^ Pos.to_nat p.
Proof.
  induction p as [q IHq | q IHq| ].
 (* Rest of proof omitted *)
```

```
Lemma Pos_bpow_ok :
  forall p x, Pos_bpow x p == x ^ Pos.to_nat p.
(* Proof omitted *)
```

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

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

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

End About_power.
```

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 naive one. Can we study *within Coq* the respective efficiency of both functions? Let us take a simple example with the exponent 17, in any `EMonoid`.

```
Eval simpl in   fun (x:A) => x ^b 17.
```

```
 = fun x : A =>
       x *
       (x * x * (x * x) * (x * x * (x * x)) *
        (x * x * (x * x) * (x * x * (x * x))))
     : A -> A
```

Therefore, we note that the term (`fun (x:A) =>x ^b 17`) is convertible, — *thus logically indistinguishable* —, with a function that performs 16 multiplications.

Likewise, let us simplify the term (`fun (x:A) =>x ^ 17`):

```
Eval simpl in   fun x =>  x ^ 17.
```

```
= fun x : A =>
   x * (x *  (x *  (x *  (x *   (x *  (x *  (x *
    (x * (x * (x * (x * (x * (x * (x * (x * (x * one)))))
  ))))))))))
```

From these tests, we may infer that representing exponentiation algorithms as Coq functions hides information about the real structure of the computations, particularly the sharing on intermediate computations.

Thus, we propose to define a data structure that makes explicit the sequence of multiplications that lead to the computation of x^n. For instance, the values of `x * x` and `x * x * (x * x)` are used twice in the computation of x^{17} with the binary algorithm. This information should appear explicitly in the data structure chosen for representing exponentiation algorithms.

It is well known that local variables can be used to store intermediate results. In an `ISWIM - ML` style, the function computing x^{17} could be written as follows:

```
Definition pow_17  (x:A) :=
  let x2 := x * x in
  let x4 := x2 * x2 in
  let x8 := x4 * x4 in
  let x16 := x8 * x8 in
  x16 * x.
```

Unfortunately, Coq's **let-in** construct is useless for our purpose, since ζ-conversion would make the sharing of computations disappear.

```
Eval cbv  zeta beta delta [pow_17]  in  pow_17.
```

```
= fun x : A =>
      x * x * (x * x) * (x * x * (x * x)) *
      (x * x * (x * x) * (x * x * (x * x))) * x
    : A -> A
```

In the next section, we propose to use a *data structure* for representing the computations that lead to the evaluation of some power x^n, where intermediary results are explicitly named for further use in the rest of the computation.

2.5.1 Addition chains

An *addition chain* (In short : *a chain*) [Bra39] is a representation of a sequence of intermediate steps that lead to the evaluation of some x^n, under the assumption that each of these steps is a computation of a power x^i, with $i < n$.

In articles from the combinatorist community, *e.g.* [Bra39, BB87], addition chains are represented as sequences of positive integers, each member of which is either 1 or the sum of two previous elements. For instance, the three following sequences are addition chains for the exponent 87:

$$c_{87} = (1, 2, 3, 6, 7, 10, 20, 40, 80, 87) \tag{2.9}$$
$$c'_{87} = (1, 2, 3, 4, 7, 8, 16, 23, 32, 64, 87) \tag{2.10}$$
$$c''_{87} = (1, 2, 4, 8, 16, 32, 64, 80, 84, 86, 87) \tag{2.11}$$

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 graphical representations of c_{87} and c'_{87}. Please note that some chains may be represented by various different dags. For instance, we can associate four different dags to the chain $(1, 2, 3, 4, 6, 9, 13)$.

Figure 2.1: Graphical representation of c_{87} (9 multiplications)

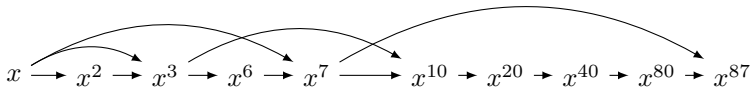
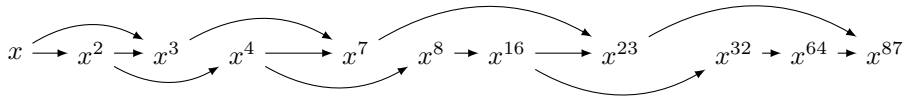


Figure 2.2: Graphical representation of c'_{87} (10 multiplications)



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 exponents (without counting the initial 1). This length is the number of multiplications needed for computing the x^is by applying the following algorithm:

> For any item i of c, there exists j and k in c, where $i = j + k$, and x^j and x^k are already computed.
>
> Thus, compute $x^i = x^j \times x^k$.

In our little example, we have $|c_{87}| = 9 < 10 = |c'_{87}|$. In the rest of this chapter, we will try to focus on the following aspects:

- Define a representation of addition chains that allows to compute efficiently x^n in any monoid, for quite large exponents n;
- Certify that our representation of chains is correct, *i.e.*, determines a computation of x^n for a given n;
- Define and certify functions for automatically generating correct and shortest as possible chains.

In a previous work [BCHM95, BCS91, Cas], addition chains were represented so as to allow efficient computations of powers and certification of a family of automatic chain generators. We present here a new implementation that takes into account some advances in the way we use Coq: generalized rewriting, type classes, parametricity, etc.

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) [Rey93] is a way to make explicit the control in the evaluation of an expression, in a purely functional way. For every intermediate computation step, the result is sent to a *continuation* that executes the further continuations. When the continuation is a lambda-abstraction, its bound variable gives a *name* to this result
- Like in Parametric Higher Order Abstract Syntax (PHOAS) [Chl08], the local variables associated to intermediate results are represented by variables of type A, where A is the underlying type of the considered monoid.

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 Module Powers.Chains

```
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* [Chl11] for more details on monadic notations in Coq.

```
Notation "z '<---'  x 'times' y ';' e2 " :=
  (Mult x y  (fun z => e2))
    (right associativity, at level 60).
```

The `computation` type family is able to express sharing of intermediate computations. For instance, the computation of 2^7 depicted in Figure 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.
```

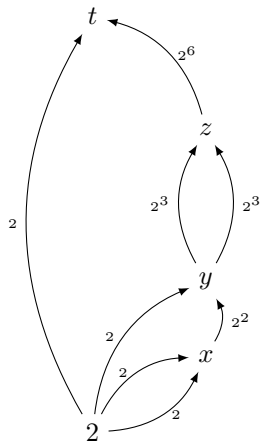


Figure 2.3: The dag associated to a computation of 2^7

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`:

```
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 computation. Note that the function `chain_length` calls the auxiliary function `computation_length`, with the variable `A` instantiated to the singleton type `unit`.

Any other type in Coq would have fitted our needs, but `unit` and its unique inhabitant `tt` was the simplest solution.

```
Fixpoint computation_length {A} (a:A)(m : @computation A)
  : nat :=
match m with
  | Mult _ _ k => S (computation_length a (k a))
  | _ => 0%nat
end.

Definition chain_length (c:chain)
   := computation_length tt (c _ tt).

Compute chain_length C87.
```

```
= 9 : nat
```

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.
```

```
 =  3232579099291745342922739807213602718533873%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 than or equal to the length of any chain correct for p.*

```
Definition optimal (p:positive) (c : chain) :=
 forall c', chain_correct p c' ->
            (chain_length c <= chain_length c')%nat.
```

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, 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 43.

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 20.

```
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[BC04] 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 35, 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 35,
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 naive 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 naive 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. Strachey [Str00] explores the nature of *parametric polymorphism*: "*Polymorphic functions behave uniformly for all types*" then Reynolds [Rey83] formalizes this notion through binary relations. Wadler [Wad89], then Cohen *et al.* [CDM13b] 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 [CDM13b]). The reader may also consult a survey by D. Brown on the most important contributions to the notion of parametricity [Bro10].

```
Lemma exponent_pos2nat : forall c: chain,  parametric c ->
  the_exponent_nat c = Pos.to_nat (the_exponent c).

Lemma exponent_pos_of_nat : forall c: chain,  parametric c ->
  the_exponent c = Pos.of_nat (the_exponent_nat c).

Lemma param_correctness (c:chain) :
   parametric c ->
   chain_correct 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 39, 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 than or equal to 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 42). 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.
```

Exercice 2.1 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 [BCHM95] for details.

Considering chain generators and their correctness, we may consider the dual of decomposition of exponents: we would like to write *composable* correct chain generators. For instance, we want to build some object that, "composed" with any correct chain for n, returns a correct chain for $3n$.

2.8.0.0.1 Note: All the Coq material described in this section is available on Module Powers/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 2.2 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.

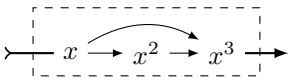


Figure 2.5: Graphical representation of `F3`

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-chains` is easily defined (see Figure 2.6).

```
Definition Fcompose (f1 f2: Fchain) : Fchain  :=
 fun   A k x =>  f1  A (fun y => f2 A k y) x.

Lemma F1_neutral_l : forall f, Fcompose F1 f = f.
Proof. reflexivity. Qed.

Lemma F1_neutral_r : forall f, Fcompose f F1 = f.
Proof. reflexivity. Qed.
```

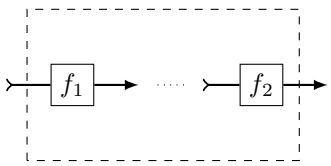


Figure 2.6: Composition of f-chains f_1 and f_2 (`Fcompose`)

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.
```

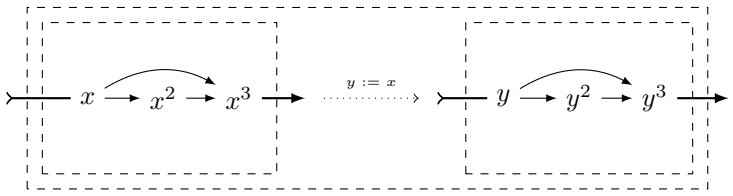


Figure 2.7: Composition of F-chains: F9

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.

2.8.2.1 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).
```

2.8.2.2 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.
```

2.8.2.3 Composition of correct f-chains: a first attempt

We are now looking for a way to generate correct chains for any positive number. It seems obvious that we could use Fcompose for building a correct f-chain for $n \times p$ by composition of a correct f-chain for n and a correct f-chain for p.

Let us try to certify this construction:

```
Module Bad2.

Lemma Fcompose_correct_attempt :
  forall f1 f2 n1 n2, Fchain_correct 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 56) 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.
```

2.8.2.4 A better definition of properness

The following generalization will allow us to consider continuations that are different (according to Leibniz equality) but lead to equivalent computations and results.

```
Definition Fkont_equiv  `(M : @EMonoid A op E_one E_equiv)
 (k k': Fkont A )  :=
 forall x y : A, x == y ->
                 computation_equiv op E_equiv  (k x)  (k' y).

Class Fchain_proper (fc : Fchain) := Fchain_proper_prf :
 forall  `(M : @EMonoid A op E_one E_equiv) k k' ,
    Fkont_proper M k -> Fkont_proper M k' ->
    Fkont_equiv M k k' ->
    forall x y, x == y ->
        @computation_equiv _ op E_equiv
                           (fc A k x)
                           (fc A k' y).
```

2.8.2.5 Examples

The definition above allows us to build simply several instances of the class `Fchain_proper`:

```
Instance F1_proper : Fchain_proper F1.
Proof.
  intros until M ; intros k k' Hk Hk' H a b H0; unfold F1; cbn;
  now apply H.
Qed.
```

```
Ltac add_op_proper M H :=
 let h := fresh H in
   generalize (@Eop_proper _ _ _ _ M); intro h.


Instance F3_proper : Fchain_proper F3.
Proof.
  intros  A op one equiv M  k k' Hk Hk'  Hkk' x y Hxy;
  apply Hkk'; add_op_proper M H; repeat rewrite Hxy;
  reflexivity.
Qed.
```

We are now able to prove `Fexp2` n's correctness by induction on n:

```
Instance Fexp2_nat_proper (n:nat) :
                           Fchain_proper (Fexp2_of_nat n).
Proof.
 induction n; cbn.
   - apply F1_proper.
   - apply Fcompose_proper ; [apply F2_proper | apply IHn].
Qed.
```

```
Lemma  Fexp2_nat_correct (n:nat) :
         Fchain_correct_nat (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 can express this feature in terms of continuations that accept two arguments. Thus, we extend our previous definitions to chains that return two different powers of their argument[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.
```

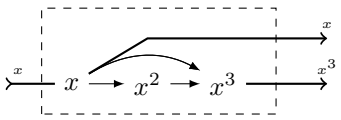


Figure 2.8: Graphical representation of `K3_1`

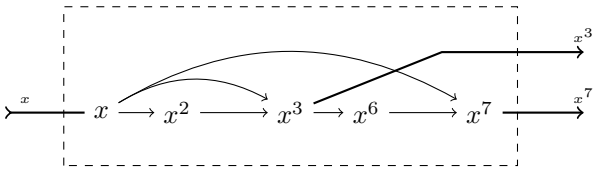


Figure 2.9: Graphical representation of `K7_3`

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) :=
```

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

```
 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`.

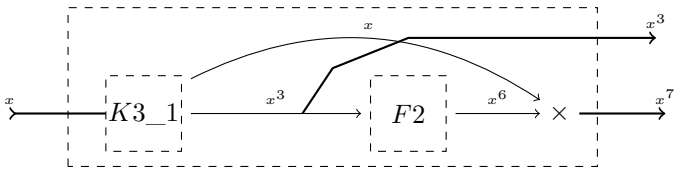


Figure 2.10: Decomposition of `K7_3`

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.

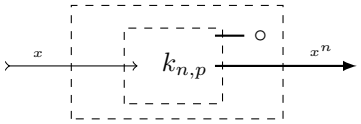


Figure 2.11: The `K2F (knp)` construction

```
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.

```
Definition KFK (kbr : Kchain) (fq : Fchain) : Kchain  :=
  fun A k a =>
    kbr A  (fun xb xr =>
              fq A (fun y =>
```

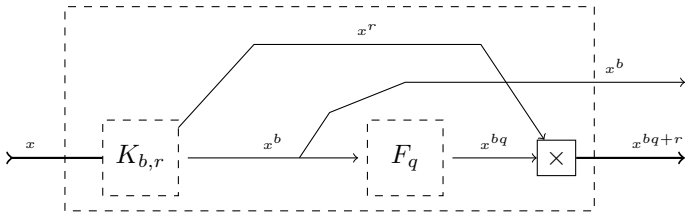


Figure 2.12: The KFK combinator

```
                    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.

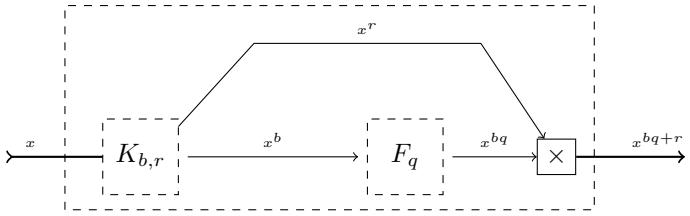


Figure 2.13: The KFF combinator

```
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}

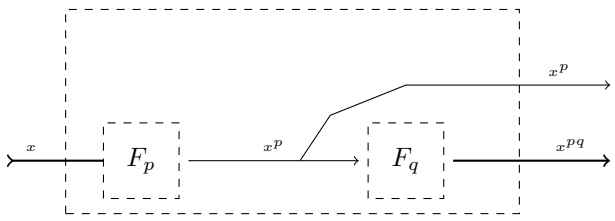


Figure 2.14: The FFK combinator

```
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 explicitly with mutual recursion, we defined a type called `signature` for representing both forms of function calls. Thus, it is easy to define a decreasing measure on type `signature` for proving termination. Likewise, correctness and properness statements are also indexed by this type.

```
Inductive signature : Type :=
|  (** Fchain for the exponent n *)
     gen_F (n:positive)
| (** Kchain for the exponents p+d  and p *)
     gen_K (p d: positive).
```

The following dependently-typed functions will help us to specify formally any correct chain generator.

```
(**
  exponent associated with a signature:
*)
Definition signature_exponent (s:signature) : positive :=
 match s with
| gen_F n => n
| gen_K p d  =>  p + d
end.
```

```
(**
Type of the associated continuation
*)

Definition kont_type (s: signature)(A:Type) : Type :=
match s with
| gen_F _  => Fkont A
| gen_K _ _   => Kkont A
end.

Definition chain_type (s: signature) : Type :=
 match s with
| gen_F _   => Fchain
|  gen_K _ _  => Kchain
end.
```

```
Definition correctness_statement (s: signature) :
chain_type s -> Prop :=
match s  with
  | gen_F p => fun ch => Fchain_correct 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 [BCHM95], we decompose n into $bq + r$, where $1 < b < n$, and compose the recursively built chains for q and r on one side, and q on the other side.

The efficiency of this method depends on the choice of b. In [BCHM95], the function that maps n into b is called a *strategy*. 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 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 9 sub-goals, for proving that the measure on signatures is strictly decreasing along the recursive calls. They are solved with the help of Standard Library's lemmas on arithemtic of `positive` numebers end Euclidean division.

```
Function chain_gen  (s:signature) {measure signature_measure}
:  chain_type s :=
  match s  return chain_type s with
    | gen_F i =>
      if pos_eq_dec i 1 then F1 else
        if pos_eq_dec i 3
        then F3
        else
          match exact_log2 i with
              Some p => Fexp2 p
            | _ =>
              match N.pos_div_eucl i (Npos (dicho i))
              with
                | (q, 0%N) =>
                  Fcompose  (chain_gen (gen_F (dicho i)))
                            (chain_gen (gen_F (N2Pos q)))
                | (q,r)  => KFF (chain_gen
                                   (gen_K (N2Pos r)
                                          (dicho i - N2Pos r)))
                                (chain_gen (gen_F (N2Pos q)))

              end end
```

```
    | gen_K p d =>
      if pos_eq_dec p 1 then FK (chain_gen (gen_F (1 + d)))
      else
        match N.pos_div_eucl (p + d)  (Npos p) with
          | (q, 0%N) => FFK   (chain_gen (gen_F p))
                              (chain_gen (gen_F (N2Pos q)))
          | (q,r)  => KFK (chain_gen (gen_K (N2Pos r)
                                            (p - N2Pos r)))
                          (chain_gen (gen_F (N2Pos q)))
        end
  end.
(* A lot of arithmetic proofs omitted *)
Defined.

Definition make_chain (n:positive) : chain :=
 F2C (chain_gen (gen_F n)).
```

Thanks to the `Recdef` package, we are now able to get automatically built chains using the dichotomic strategy.

```
Compute make_chain 87.
```

```
=  fun (A : Type) (x : A) =>
      x0 <--- x times x;
      x1 <--- x0 times x;
      x2 <--- x1 times x1;
      x3 <--- x2 times x;
      x4 <--- x3 times x1;
      x5 <--- x4 times x4;
      x6 <--- x5 times x5;
      x7 <--- x6 times x6;
      x8 <--- x7 times x3;
      Return x8
    : chain
```

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.

```
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 than or equal to $\lfloor \log_2 n \rfloor$.
3. Prove that, for any positive n, the length of any Euclidean chain generated by the dichotomic strategy is always less than or equal to the length of `binary_chain` n, and for an infinite number of positive integers n, the first chain is strictly shorter than the latter.
4. Prove that our implementation of the dichotomic strategy describes the same function as in the litterature (for instance [BCHM95].) This is important if we want to follow the complexity analyses in this and similar articles.
5. Study how to *compile* a chain into imperative code, using a register allocation strategy (it may be useful to define *chain width*).

Remark: The first two questions of the list above should involve a universal quantification on type `chain`. It may be necessary (but we're not sure) to consider some restriction on parametric chains.

Appendices

Bibliography

[Abr96] J.-R. Abrial. *The B-book: Assigning Programs to Meanings*. Cambridge University Press, New York, NY, USA, 1996.

[Abr10] Jean-Raymond Abrial. *Modeling in Event-B: System and Software Engineering*. Cambridge University Press, New York, NY, USA, 1st edition, 2010.

[BB87] Jean Berstel and Srecko Brlek. On the length of word chains. *Inf. Process. Lett.*, 26(1):23–28, 1987.

[BC04] Yves Bertot and Pierre Castéran. *Interactive Theorem Proving and Program Development. Coq'Art: The Calculus of Inductive Constructions*. Springer, 2004. `http://www.labri.fr/perso/casteran/CoqArt/index.html`.

[BCHM95] Srecko Brlek, Pierre Castéran, Laurent Habsieger, and Richard Mallette. On-line evaluation of powers using euclid's algorithm. *ITA*, 29(5):431–450, 1995.

[BCS91] 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.

[Bra39] Alfred Brauer. On addition chains. *Bull. Amer. Math. Soc.*, 45(10):736–739, 10 1939.

[Bro10] Daniel Brown. Parametricity. http://www.ccs.neu.edu/home/matthias/369-s10/Transcript/parametricity.pdf, 2010. Available on Matthias Felleisen page.

[Bur75] William H. Burge. *Recursive programming techniques / William H. Burge*. Addison-Wesley Pub. Co Reading, Mass, 1975.

[Cas] Pierre Castéran. Additions. User Contributions to the Coq Proof Assistant. https://github.com/coq-contribs/additions.

[CDM13a] Cyril Cohen, Maxime Dénès, and Anders Mörtberg. Refinements for Free! In *Certified Programs and Proofs*, pages 147 – 162, Melbourne, Australia, December 2013.

[CDM13b] 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.

[Chl08] 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.

[Chl11] Adam Chlipala. *Certified Programming with Dependent Types*. MIT Press, 2011. `http://adam.chlipala.net/cpdt/`.

[Coq] Coq Development Team. The coq proof assistant. https://coq.inria.fr.

[CS] 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.

[Gon08] Georges Gonthier. Formal proof — the four-color theorem. *Notices of the American Mathematical Society*, 55(11), December 2008.

[H+15] Thomas Hales et al. A formal proof of the Kepler conjecture. https://arxiv.org/abs/1501.02155, 2015.

[P+] Benjamin Pierce et al. Software foundations. https://softwarefoundations.cis.upenn.edu/.

[Rey83] 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).

[Rey93] John C. Reynolds. The discovery of continuations. *Lisp and Symbolic Computation*, 6:233–247, 1993.

[SO08] 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.

[Str00] Christopher Strachey. Fundamental concepts in programming languages. *Higher Order Symbol. Comput.*, 13(1-2):11–49, April 2000.

[SvdW11] Bas Spitters and Eelis van der Weegen. Type classes for mathematics in type theory. http://arxiv.org/pdf/1102.1323.pdf, 2011.

[Wad89] 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.

2.10 How to install the libraries

- The present distribution has been checked with version 8.12.0 of the Coq proof assistant, with the plug-ins `coq-paramcoq` and `coq-equations`.
- just go into the top directory, and type "`make`"

Index