16

* Proof by Reflection

Proof by reflection is a characteristic feature of proving in type theory. There is a programming language embedded inside the logical language and it can be used to describe decision procedures or systematic reasoning methods. We already know that programming in *Coq* is a costly task and this approach is only worth the effort because the proof process is made much more efficient. In some cases, dozens of rewrite operations can be replaced with a few theorem applications and a convertibility test of the Calculus of Inductive Constructions. Since the computations of this programming language do not appear in the proof terms, we obtain proofs that are smaller and often quicker to check.

In this chapter, we describe the general principle and give three simple examples that involve proofs that numbers are prime and equalities between algebraic expressions.

16.1 General Presentation

Proof by reflection is already visible in the proofs that we performed in earlier chapters to reason about functions in *Coq*. To handle these functions, we often rely on term reductions: $\beta\delta\zeta$-reduction for simple functions and ι-reduction for recursive functions.

Let us have a second look at a simple proof, the proof that natural number addition is associative:

```
Theorem plus_assoc : ∀x y z:nat, x+(y+z) = x+y+z.
Proof.
 intros x y z; elim x.
```

Here, the proof by induction in the `elim` tactic produces two goals. The first one has the following shape:

```
  ...
  ============================
   0+(y+z) = 0+y+z
```

We usually call `auto` or `trivial` to solve this goal, but with a closer look, we see that these tactics actually perform the following operation:

```
exact (refl_equal (y+z)).
```

This is surprising, since neither the left-hand side nor the right-hand side of the equation is the term "`y+z`." However, both terms are *convertible* to this expression. To check this proof step, the *Coq* system must perform a small computation, leading to the replacement of all instances of "`0+`m" with m, as is expressed in the definition of `plus`. This operation is performed twice, once in the left-hand side for "$m =$ `y+z`" and once in the right-hand side for $m =$ `y`, but this is invisible to the user.

This example shows that simplifying formulas plays a significant role in the proof process, since we can replace reasoning steps with a few computation steps. Here the reasoning steps are elementary but the aim of proof by reflection is to replace complex combinations of reasoning steps with computation. In fact, we use reduction to execute decision procedures. In proofs by reflection, we describe explicitly inside the logical language the computations that are normally performed by automatic proof tools. We naturally have to prove that these computations do represent the reasoning steps they are supposed to represent, but like the typing process, these proofs need to be done only once. The proof tools we obtain do not have to build a new proof for each piece of input data.

There are two large classes of problems where this kind of proof technique is useful. In the first class, we consider a predicate `C:T→Prop` where `T` is a data type and we have a function `f:T→bool` such that the following theorem holds:

```
f_correct : ∀x:T, f x = true → C x.
```

If `f` is defined in such a way that we can reduce "`f` t" to `true` for a large class of expressions t, then the following proof can be used as a proof of "`C` t":

```
f_correct t (refl_equal true):C t
```

Except for the occurrence of t in the first argument of `f_correct`, the size of this proof does not depend on t. In practice, this kind of tactic applies only if t is a term without variables, in other words if t is built only with the constructors of inductive types. In the next section, we give an example of this class of problems with the verification that a given number is prime and we obtain a tactic that is much quicker than the tactic described in Sect. 7.6.2.1.

The second class of problems where computation can help is the class of algebraic proofs such as proofs relying on rewriting modulo the associativity or the commutativity of some operators. For these proofs, we again consider a type T and we exhibit an "abstract" type A, with two functions $i : A \to T$ and $f : A \to A$. The function i is an interpretation function that we can use to associate terms in the concrete type T with abstract terms. The function f reasons on the abstract terms. The reflection process relies on a theorem that

expresses that the function f does not change the value of the interpreted term:

```
f_ident : ∀x:A, i (f x)= i x
```

Thus, to prove that two terms t_1 and t_2 are equal in T, we only need to show that they are the images of two terms a_1 and a_2 in A such that "$f\ a_1{=}f\ a_2$." We give an example of this kind of algebraic reasoning by reflection in the third section of this chapter where we study proofs of equality based on associativity. We then show an elaboration of this method to study proofs of equality based on associativity and commutativity.

16.2 Direct Computation Proofs

The functions used in proof by reflection are particular: it is important that $\beta\delta\zeta\iota$-reductions as performed by the tactics `simpl`, `lazy`, `cbv`, or `compute` are enough to transform the expression into an appropriate form. In practice, all the functions have to be programmed in a structural recursive way, sometimes by relying on the technique of bounded recursion described in Sect. 15.1.

Complexity matters. The functions are executed in the proof system, using the internal reduction mechanisms, whose efficiency compares poorly with the efficiency of conventionally compiled programming languages. For algorithms that will be used extensively, it is worth investing in the development and the formal proof of efficient algorithms.

For instance, we are interested in the *Coq* proof that a reasonably sized natural number is prime:[1] we need to show that this number cannot be divided by a large collection of other numbers. Divisions by successive subtractions are rather inefficient and it is better to convert natural numbers to integers, which use a binary representation, before performing all divisibility tests.

Computing Remainders

The *Coq* library provides a function for Euclidean division in the module `Zdiv`:

```
Require Export Zdiv.
```

The division function is called `Zdiv_eucl` and has type `Z→Z→Z*Z`. This is a weak specification, but the companion theorem gives more suitable information:

```
Z_ div_ mod : ∀ a b:Z, (b > 0)%Z →
        let (q, r) := Zdiv_ eucl a b in
        a = (b*q+r)%Z ∧ (0 ≤ r < b)%Z
```

The module `Zdiv` also provides a function `Zmod` that only returns the second element of the pair.

[1] This example was suggested by M. Oostdijk and H. Geuvers.

Setting Up Reflexion

We need a first theorem that relates the existence of a divisor with remainders using integer division. We do not detail the proof here and only rely on this property as an axiom:

```
Axiom verif_divide :
    ∀m p:nat, 0 < m → 0 < p →
    (∃q:nat, m = q*p)→(Z_of_nat m mod Z_of_nat p = 0)%Z.
```

Our intention is to check that a number is prime by verifying that the division using every number smaller than it produces a non-zero remainder. Here is a justification that only smaller numbers need to be checked, also accepted as an axiom:

```
Axiom divisor_smaller :
    ∀m p:nat, 0 < m → ∀q:nat, m = q*p → q ≤ m.
```

We can write a function that tests the result of division for all smaller numbers.

```
Fixpoint check_range (v:Z)(r:nat)(sr:Z){struct r} : bool :=
  match r with
    0 ⇒ true
  | S r' ⇒
    match (v mod sr)%Z with
      Z0 ⇒ false
    | _ ⇒ check_range v r' (Zpred sr)
    end
  end.

Definition check_primality (n:nat) :=
  check_range (Z_of_nat n)(pred (pred n))(Z_of_nat (pred n)).
```

We can test this function on a few values:

```
Eval compute in (check_primality 2333).
    = true : bool

Eval compute in (check_primality 2330).
    = false : bool
```

It looks simpler to write this function with only two arguments, as in the following definition, but this function redoes the conversion of the divisor at every step.

```
Fixpoint check_range' (v:Z)(r:nat){struct r} : bool :=
  match r with
    0 ⇒ true | 1 ⇒ true
  | S r' ⇒
```

```
      match (v mod Z_of_nat r)%Z with
      | 0%Z ⇒ false
      | _ ⇒ check_range' v r'
      end
  end.

Definition check_primality' (n:nat) :=
  check_range' (Zpos (P_of_succ_nat (pred n)))(pred (pred n)).
```

This variant is much slower. Each call to the function `inject_nat` has a linear cost in the number being represented. In the function `check_range` this computation is avoided and replaced by a subtraction on a binary number at each step. This subtraction has a linear cost in the size of the binary representation, but this binary representation only has a logarithmic size. The cost is much lower. It is often convenient to test the complexity and the validity of the function before starting to prove that it is correct; here a few experiments were enough to establish that the function `check_range` was better than the function `check_range'`.

We can now prove the theorems that show that our functions are correct. We use two results, which we take as axioms here, but which should be proved in a regular development:

```
Axiom check_range_correct :
  ∀(v:Z)(r:nat)(rz:Z),
  (0 < v)%Z →
  Z_of_nat (S r) = rz →
  check_range v r rz = true →
  ∼(∃k:nat, k ≤ S r ∧ k ≠ 1 ∧
                      (∃q:nat, Zabs_nat v = q*k)).

Axiom check_correct :
  ∀p:nat, 0 < p → check_primality p = true →
  ∼(∃k:nat, k ≠ 1 ∧ k ≠ p ∧ (∃q:nat, p = q*k)).
```

Proving that an arbitrary number is prime can now be done with a simple tactic, as in the following example:

```
Theorem prime_2333 :
 ∼(∃k:nat, k ≠ 1 ∧ k ≠ 2333 ∧ (∃q:nat, 2333 = q*k)).
Time apply check_correct; auto with arith.
Proof completed.
Finished transaction in 132. secs (131.01u,0.62s)
Time Qed.
...
Finished transaction in 59. secs (56.79u,0.4s)
```

This proof takes a few minutes (adding up the time for building and checking the proof term), while the naïve procedure described in Sect. 7.6.2.1 was

unable to cope with a number this size. There are various simple ways to improve our development. The first is to test only odd divisors and use pattern matching to check that the number is not even; the second is to limit tests to numbers that are smaller than the square root (the module `ZArith` provides a square computation function called `Zsqrt`).

Oostdijk developed an even more elaborate tactic [20] based on a lemma by Pocklington, which can cope with numbers that are written with several dozens of digits in decimal representation.

Exercise 16.1 *** Prove the lemmas* `verif_divide`*,* `divisor_smaller`*,* `check_range_correct`*, and* `check_correct`*.*

Exercise 16.2 *** Show that when a number n is the product of two numbers p and q, then one of these numbers is smaller than the square root of n. Use this lemma to justify a method by reflection that only verifies odd divisors smaller than the square root.*

16.3 ** Proof by Algebraic Computation

Reduction can be used to compute on other things than just numbers. In this section, we study examples where we compute symbolically on algebraic objects.

16.3.1 Proofs Modulo Associativity

As an illustrative example, we study proofs that two expressions of type `nat` are equal when these proofs only involve associativity of addition. We call them proofs of equality modulo associativity.

The expressions of type `nat` that we consider are binary trees, where the nodes are not labeled and represent the `plus` function, while the leaves are labeled by values that represent arbitrary arithmetic expressions. Intuitively, proofs of equality modulo associativity are done by forgetting the parentheses associated to all additions on the two sides of the equation and then verifying that the same expressions appear on both sides in the same order. Thus, it is obvious that the expressions

$$x + ((y + z) + w) \qquad \text{and} \qquad (x + y) + (z + w)$$

are equal.

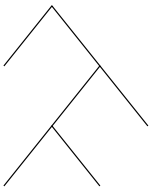
Without using reflection, this kind of proof can be done with a one-line combined tactic, as in the following example:

```
Theorem reflection_test :
 ∀x y z t u:nat, x+(y+z+(t+u)) = x+y+(z+(t+u)).
Proof.
```

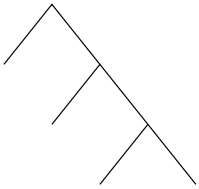
```
 intros; repeat rewrite plus_assoc; auto.
Qed.
```

The tactic used in this example is independent of the terms in the equality, but the proof it builds increases as the expressions do because of the `repeat` tactical. In practice, this means that the time taken by this tactic increases quickly with the size of the expressions. The time taken by the `Qed` command also increases badly.

Forgetting the parentheses related to additions in an expression is the same thing as rewriting with the associativity theorem until all additions are pushed on the right-hand side of other additions. Graphically, this corresponds to transforming a tree with the form



into a tree with the form



We want to describe a function that transforms, for instance, the expression $x + ((y + z) + w)$ into $x + (y + (z + w))$, but this function cannot be defined using a pattern matching construct where the function `plus` plays a special role, because `plus` is not a constructor of the type `nat` and it is meaningless to ask whether any `x` could be the result of an addition. The type `nat` is not suited for this purpose, but we can define our function as a function working on an abstract type of binary trees that we define with the following command:

```
Inductive bin : Set := node : bin→bin→bin | leaf : nat→bin.
```

We can first define a function that reorganizes a tree to the right, where the tree that must appear in the rightmost position is given as an argument, and then give a function that simply performs the whole processing:

```
Fixpoint flatten_aux (t fin:bin){struct t} : bin :=
  match t with
  | node t1 t2 ⇒ flatten_aux t1 (flatten_aux t2 fin)
  | x ⇒ node x fin
  end.
```

```
Fixpoint flatten (t:bin) : bin :=
  match t with
  | node t1 t2 ⇒ flatten_aux t1 (flatten t2)
  | x ⇒ x
  end.
```

This function can be tested directly in *Coq* to check that it really produces trees where no addition has another addition as its first argument:

```
Eval compute in
  (flatten
    (node (leaf 1) (node (node (leaf 2)(leaf 3)) (leaf 4)))).
```
= node (leaf 1) (node (leaf 2) (node (leaf 3) (leaf 4))) : bin

The next step is to show how binary trees represent expressions of type `nat`, with the help of an *interpretation function*:

```
Fixpoint bin_nat (t:bin) : nat :=
  match t with
  | node t1 t2 ⇒ bin_nat t1 + bin_nat t2
  | leaf n ⇒ n
  end.
```

This interpretation function clearly states that the operator `node` represents additions. Here is a test of this function:

```
Eval lazy beta iota delta [bin_nat] in
 (bin_nat
   (node (leaf 1) (node (node (leaf 2) (leaf 3)) (leaf 4)))).
```
= 1+(2+3+4) : nat

The main theorem is that changing the shape of the tree does not change the value being represented. We start with a lemma concerning the auxiliary function `flatten_aux`. Intuitively, it should represent an addition:

```
Theorem flatten_aux_valid :
 ∀t t':bin, bin_nat t + bin_nat t' = bin_nat (flatten_aux t t').
```

The proof of this lemma follows the structure of the function `flatten_aux`, but we do not detail it here. Nevertheless, it is important to know that the theorem of associativity of addition plays a role in this proof. This lemma is used for the next theorem, which concerns the main function `flatten`:

```
Theorem flatten_valid : ∀t:bin, bin_nat t = bin_nat (flatten t).
```

We can obtain a corollary where `flatten_valid` is applied on both sides of an equation:

```
Theorem flatten_valid_2 :
  ∀t t':bin, bin_nat (flatten t) = bin_nat (flatten t')→
  bin_nat t = bin_nat t'.
Proof.
 intros; rewrite (flatten_valid t); rewrite (flatten_valid t');
 auto.
Qed.
```

We now have all the ingredients to perform a proof that $x+((y+z)+w)$ and $(x+y)+(z+w)$ are equal modulo associativity:

```
Theorem reflection_test' :
 ∀x y z t u:nat, x+(y+z+(t+u))=x+y+(z+(t+u)).
Proof.
 intros.
 change
   (bin_nat
      (node (leaf x)
         (node (node (leaf y) (leaf z))
               (node (leaf t)(leaf u)))) =
    bin_nat
      (node (node (leaf x)(leaf y))
         (node (leaf z)(node (leaf t)(leaf u))))).
 apply flatten_valid_2; auto.
Qed.
```

This proof involves only two theorems: `flatten_valid_2` and the reflexivity of equality. Nevertheless, there is a tedious step, where the user must give the `change` tactic's argument by hand. This step can also be automated with the help of the $\mathcal{L}$tac language described in Sect. 7.6:

```
Ltac model v :=
  match v with
  | (?X1 + ?X2) ⇒
    let r1 := model X1 with r2 := model X2 in
    constr:(node r1 r2)
  | ?X1 ⇒ constr:(leaf X1)
  end.

Ltac assoc_eq_nat :=
  match goal with
  | [ |- (?X1 = ?X2 :>nat) ] ⇒
   let term1 := model X1 with term2 := model X2 in
   (change (bin_nat term1 = bin_nat term2);
    apply flatten_valid_2;
    lazy beta iota zeta delta [flatten flatten_aux bin_nat];
    auto)
```

```
  end.
```

The function `model` needs to construct a term that is obtained by applying a function of the *Gallina* language to an expression of the *Gallina* language and this application should not be confused with the application of the $\mathcal{L}$tac language. To make the distinction clear, the *Coq* system imposes that we mark this application with a prefix `constr:` (this means a term of the Calculus of Constructions).

The tactic `assoc_eq_nat` summarizes in a single keyword the whole collection of steps to prove the equation using this method. Here is a sample session:

```
Theorem reflection_test'' :
 ∀x y z t u:nat, x+(y+z+(t+u)) = x+y+(z+(t+u)).
Proof.
 intros; assoc_eq_nat.
Qed.
```

We advise the reader to test this tactic on a larger collection of cases.

Exercise 16.3 *Prove the theorems* `flatten_aux_valid`*,* `flatten_valid`*, and* `flatten_valid_2`*.*

16.3.2 Making the Type and the Operator More Generic

The technique described in the previous section should be reusable in all cases where a binary operation is associative. For natural numbers, it should also apply for multiplication, but for other types, we should also be able to use it for addition and multiplication of integers, real numbers, rational numbers, and so on. We now describe how to make our tactic more general, by abstracting over both the type and the operator. We use the section mechanism as it is provided in *Coq*:

```
Section assoc_eq.
```

We can describe the various elements that should vary for different uses of the tactic. We must have a data type (a type in the `Set` sort), a binary operation in this type, and a theorem expressing that this operation is associative:

```
Variables (A : Set)(f : A→A→A)
  (assoc : ∀x y z:A, f x (f y z) = f (f x y) z).
```

We now must build a function mapping terms of type `bin` to terms of type `A`. Here we have to be careful because the leaves of the type `bin` are labeled with values of type `nat` rather than values of type `A`. We could change the `bin` structure to be polymorphic so that leaves do contain values of type `A`, but we prefer to keep using labels of type `nat` and to add a list of values of type `A` as an argument to the interpretation function. This choice later turns

out to be useful to handle commutativity. To interpret natural numbers with respect to a list of values, we need a function `nth` with three arguments: a natural number, a list of terms of the type `A`, and a default value of the type `A` that is used to interpret natural numbers that are larger than the length of the list. This function `nth` is actually provided in the *Coq* library (module `List`). The interpretation function that we present now is still closely related to the function `bin_nat`:

```
Fixpoint bin_A (l:list A)(def:A)(t:bin){struct t} : A :=
  match t with
  | node t1 t2 ⇒ f (bin_A l def t1)(bin_A l def t2)
  | leaf n ⇒ nth n l def
  end.
```

The validity theorems must also be transposed. We only give their statements and do not detail the proofs:

```
Theorem flatten_aux_valid_A :
 ∀(l:list A)(def:A)(t t':bin),
 f (bin_A l def t)(bin_A l def t') =
       bin_A l def (flatten_aux t t').

Theorem flatten_valid_A :
 ∀(l:list A)(def:A)(t:bin),
   bin_A l def t = bin_A l def (flatten t).

Theorem flatten_valid_A_2 :
 ∀(t t':bin)(l:list A)(def:A),
   bin_A l def (flatten t) = bin_A l def (flatten t')→
   bin_A l def t = bin_A l def t'.
```

We can now close the section and obtain generic theorems:

```
End assoc_eq.
Check flatten_valid_A_2.
flatten_valid_A_2:
 ∀(A:Set)(f:A→A→A),
 (∀x y z:A, f x (f y z) = f (f x y) z)→
 ∀(t t':bin)(l:list A)(def:A),
 bin_A A f l def (flatten t) = bin_A A f l def (flatten t')→
 bin_A A f l def t = bin_A A f l def t'
```

The meta-functions needed for the generic tactic are more complex, because we have to build a list of terms and to verify when a term is already present in this list. The function `term_list` simply traverses a term and collects the leaves of the binary trees whose nodes are the operation we study. The function `compute_rank` takes one of these leaves and finds the position it has in the list (making a deliberate mistake if the leaf cannot be found in the list, something

that should never happen). The function `model_aux` takes a list of leaves `l`, the studied operator, and a concrete term `v` and finds the abstract term of the type `bin` that is the inverse image of `v` by the interpretation function "`bin_A l`." The function `model_A` and the tactic `assoc_eq` combine all the auxiliary functions in the same way as before:

```
Ltac term_list f l v :=
  match v with
  | (f ?X1 ?X2) ⇒
    let l1 := term_list f l X2 in term_list f l1 X1
  | ?X1 ⇒ constr:(cons X1 l)
  end.

Ltac compute_rank l n v :=
  match l with
  | (cons ?X1 ?X2) ⇒
    let tl := constr:X2 in
    match constr:(X1 = v) with
    | (?X1 = ?X1) ⇒ n
    | _ ⇒ compute_rank tl (S n) v
    end
  end.

Ltac model_aux l f v :=
  match v with
  | (f ?X1 ?X2) ⇒
    let r1 := model_aux l f X1 with r2 := model_aux l f X2 in
      constr:(node r1 r2)
  | ?X1 ⇒ let n := compute_rank l 0 X1 in constr:(leaf n)
  | _ ⇒ constr:(leaf 0)
  end.

Ltac model_A A f def v :=
  let l := term_list f (nil (A:=A)) v in
  let t := model_aux l f v in
  constr:(bin_A A f l def t).

Ltac assoc_eq A f assoc_thm :=
  match goal with
  | [ |- (@eq A ?X1 ?X2) ] ⇒
  let term1 := model_A A f X1 X1
  with term2 := model_A A f X1 X2 in
  (change (term1 = term2);
   apply flatten_valid_A_2 with (1 := assoc_thm); auto)
  end.
```

The tactic `assoc_eq` can be used in the same way as the tactic `assoc_eq_nat` but we have to indicate which type, which binary operation, and which associativity theorem are used. Here is an example with integer multiplication:

```
Theorem reflection_test3 :
 ∀x y z t u:Z, (x*(y*z*(t*u)) = x*y*(z*(t*u)))%Z.
Proof.
 intros; assoc_eq Z Zmult Zmult_assoc.
Qed.
```

Exercise 16.4 *Using the hypothesis `f_assoc`, prove the three theorems `flatten_aux_valid_A`, `flatten_valid_A`, and `flatten_valid_A_2`.*

Exercise 16.5 *Adapt the tactic to the case where the binary operation has a neutral element, like zero for addition. It should be able to prove equalities of the form "`(x+0)+(y+(z+0))=x+(y+(z+0))`."*

16.3.3 *** Commutativity: Sorting Variables

When storing data in a list, as we did in the previous section, one establishes an order between values. This order is arbitrary, but it can be useful. An example is the case where we wish to prove equalities modulo associativity *and* commutativity. In this case, we not only want to reshape the expression, but also want to put all leaves in the same order, so that expressions that appear on both sides also appear in the same position. The development we present is directly inspired by the development of the tactic `field` by D. Delahaye and M. Mayero.

We still work with only one binary operation with algebraic properties and we use the same data type `bin` to model the expressions. Our approach first reshapes the tree so that all left-hand-side terms of additions are leaves. Thus, the tree actually looks like a list. The next step is simply to sort this list with respect to the order provided by the list storage.

The procedure we provide relies on insertion sort. We need a function that can compare two leaves with respect to the numbers in these leaves. We use the following simple structural recursive function that really reduces to `true` or `false` whenever its arguments are non-variable natural numbers.

```
Fixpoint nat_le_bool (n m:nat){struct m} : bool :=
  match n, m with
  | O, _ ⇒ true
  | S _, O ⇒ false
  | S n, S m ⇒ nat_le_bool n m
  end.
```

When sorting, we need to insert leaves in the trees representing lists that are already sorted. In the following insertion function the leaf is actually represented by a natural number, the value that should be in the leaf. In this insertion function, we consider that the tree must be "well-formed" in the sense that its left-hand side should be a leaf. If the tree is not well-formed, the leaf is inserted as a new first left subtree, without checking that the insertion appears in the right place. The base case is also particular, because there is no representation for an empty list.

```
Fixpoint insert_bin (n:nat)(t:bin){struct t} : bin :=
  match t with
  | leaf m ⇒ match nat_le_bool n m with
              | true ⇒ node (leaf n)(leaf m)
              | false ⇒ node (leaf m)(leaf n)
              end
  | node (leaf m) t' ⇒
    match nat_le_bool n m with
    | true ⇒ node (leaf n) t
    | false ⇒ node (leaf m)(insert_bin n t')
    end
  | t ⇒ node (leaf n) t
  end.
```

With this insertion function, we can now build a sorting function:

```
Fixpoint sort_bin (t:bin) : bin :=
  match t with
  | node (leaf n) t' ⇒ insert_bin n (sort_bin t')
  | t ⇒ t
  end.
```

We have to prove that this sorting function does not change the value of the expression represented by the tree. This proof relies on the assumptions that the function is associative and commutative.

```
Section commut_eq.
 Variables (A : Set)(f : A→A→A).
 Hypothesis comm : ∀x y:A, f x y = f y x.
 Hypothesis assoc : ∀x y z:A, f x (f y z) = f (f x y) z.
```

We can reuse the functions `flatten_aux`, `flatten`, `bin_A`, and the theorem `flatten_valid_A_2` (see Exercice 16.3). We have to prove that the sorting operation preserves the interpretation. A first lemma considers the insertion operation, which can also be interpreted as the binary operation being considered:

```
Theorem insert_is_f :
 ∀(l:list A)(def:A)(n:nat)(t:bin),
  bin_A l def (insert_bin n t) = f (nth n l def)(bin_A l def t).
```

With this theorem, it is easy to prove the right theorem for sorting:

```
Theorem sort_eq : ∀(l:list A)(def:A)(t:bin),
    bin_A l def (sort_bin t) = bin_A l def t.
```

As in the previous section, we also describe a theorem that applies the sorting operation on both sides of an equality:

```
Theorem sort_eq_2 :
 ∀(l:list A)(def:A)(t1 t2:bin),
   bin_A l def (sort_bin t1) = bin_A l def (sort_bin t2)→
   bin_A l def t1 = bin_A l def t2.
```

The section can now be closed. This makes the theorems more general.

```
End commut_eq.
```

The tactic that uses these theorems has the same structure as the tactic for associativity described in the previous section. Note that the theorem `sort_eq_2` is applied after the theorem `flatten_valid_A_2` so that the function `sort_bin` is only used on "well-formed" trees (the left-hand-side subterms of `bin` trees always are leaves).

```
Ltac comm_eq A f assoc_thm comm_thm :=
  match goal with
  | [ |- (?X1 = ?X2 :>A) ] ⇒
    let l := term_list f (nil (A:=A)) X1 in
    let term1 := model_aux l f X1
    with term2 := model_aux l f X2 in
    (change (bin_A A f l X1 term1 = bin_A A f l X1 term2);
      apply flatten_valid_A_2 with (1 := assoc_thm);
      apply sort_eq_2 with (1 := comm_thm)(2 := assoc_thm);
      auto)
  end.
```

Here is an example where this tactic is used:

```
Theorem reflection_test4 : ∀x y z:Z, (x+(y+z) = (z+x)+y)%Z.
Proof.
 intros x y z. comm_eq Z Zplus Zplus_assoc Zplus_comm.
Qed.
```

Exercise 16.6 *Prove `insert_is_f`, `sort_eq`, and `sort_eq_2`.*

16.4 Conclusion

The *Coq* libraries provide other more elaborate examples. The tactics `ring` and `field` are based on this technique and we advise prospective tactic developers to study these tactics and use them as inspiration.

In reflection tactics, efficiency considerations are important, because these tactics are executed inside the *Coq* logical engine, which is slower than conventional programming languages. For a tactic that is used intensively, it is worth the effort to use more efficient sorting algorithms than insertion sort and more efficient storage than lists. For instance, we could store data in binary trees, using numbers of type `positive` to denote positions. This kind of storage also provides an order on positions and the fetching operation is more efficient than looking up in a list structure.

Exercise 16.7 *** Using the notion of permutations defined in Exercise 8.4 page 216 and the counting function defined in Exercise 9.5 page 256, show that if a list is a permutation of another list, then any natural number occurs as many times in both lists.*

Build a specialized reflection tactic "`NoPerm`" that solves goals of the form "$\sim$*`perm` l l'" by finding an element of the first list that does not occur the same number of times in both lists.*