14

** Foundations of Inductive Types

14.1 Formation Rules

There is a lot of freedom in inductive definitions. A type may be a constant whose type is one of the sorts in the system, it may also be a function, this function may have a dependent type, and some of its arguments may be parameters. The constructors may be constants or functions, possibly with a dependent type, their arguments may or may not be in the inductive type, and these arguments may themselves be functions. In this section, we want to study the limits of this freedom.

14.1.1 The Inductive Type

Defining an inductive type adds to the context a new constant or function whose final type is one of the sorts `Set`, `Prop`, or `Type`.

When the constant or function describing the inductive type takes one or more arguments, we have to distinguish the *parametric* arguments from the regular arguments. The parametric arguments are the ones that appear between parentheses before the colon character, "`:`".

If an inductive type definition has a parameter, this parameter's scope extends over the whole inductive definition. This parameter can appear in the the type arguments that come after, in the type's type, and in the constructors' type. For instance, the definition of polymorphic lists, as given in the module `List` of the *Coq* library (see Sect. 6.4.1), has a parameter `A`:

```
Set Implicit Arguments.

Inductive list (A:Set) : Set :=
  nil : list A
| cons : A → list A → list A.

Implicit Arguments nil [A].
```

The parameter declaration `(A:Set)` introduces the type `A` that can be reused in the type descriptions for the two constructors `nil` and `cons`. Actually, the declared type for these constructors is not exactly the type they have after the declaration. For instance, if we check the type of `nil` we get the following answer (we need to prefix the function name with `@` to overrule the implicit argument declaration):

```
Check (@nil).
```
@nil : ∀ A:Set, list A

In practice, the type of each constructor is the type given in the inductive definition, prefixed with the parameters of the inductive types.

When using parametric arguments, one must respect a stability constraint: *the parameters must be reused exactly as in the parameter declaration wherever the inductive type occurs inside the inductive definition.* In the example of polymorphic lists, this imposes that `list` is always applied to `A` in the second and third lines of the definition. When the constraint is not fulfilled, the *Coq* system complains with an explicit error message. Here is an example:

```
Inductive T (A:Set) : Set := c : ∀B:Set, B → T B.
```
Error: The 1st argument of ''T'' must be ''A'' in ''∀ B:Set, B → T B''

When an inductive type takes arguments that should change between uses inside the inductive definition, the arguments must appear in the type declaration outside the parameter declaration. This is shown in the definition of fixed-height trees from Sect. 6.5.2, with the definition of `htree`:

```
Inductive htree (A:Set) : nat→Set :=
  hleaf : A → htree A 0
| hnode : ∀n:nat, A → htree A n → htree A n → htree A (S n).
```

Here the type `htree` takes two arguments: the first is a parameter and the second is a regular argument. The constraint that the parameter is reused without change is satisfied and the regular argument takes three different values in the four places where the inductive type occurs.

In the two examples above, the parameter is a type, but this is not necessary. We can define inductive types where parameters are plain data, as we saw in the inductive definition of equality (Sect. 8.2.6) or in strong specifications for functions (Sects. 6.5.1 and 9.4.2).

It seems that it is always possible to build an inductive type without parameters from an inductive type with parameters by "downgrading" the parameters to regular arguments. This operation is rarely interesting, because the induction principles are simpler when using parameters. For instance, here is an alternative definition of polymorphic lists:

```
Inductive list' : Type→Type :=
  nil' : ∀A:Type, list' A
| cons' : ∀A:Type, A → list' A → list' A.
```

This definition has the following induction principle:

```
Check list'_ind.
```
list'_ind
: ∀ P:∀ T:Type, list' T → Prop,
(∀ A:Type, P A (nil' A))→
(∀ (A:Type)(a:A)(l:list' A), P A l → P A (cons' A a l))→
∀ (T:Type)(l:list' T), P T l

This induction principle is more complex because it quantifies over a dependently typed predicate with two arguments instead of quantifying over a property with one argument.

14.1.2 The Constructors

14.1.2.1 Head Type Constraints

The constructors of an inductive type T are functions whose final type must be T (when the type is constant) or an application of T to arguments (when T is a function). This constraint is easy to recognize syntactically; the type has the following form:

$$t_1 \rightarrow t_2 \rightarrow \cdots \rightarrow t_l \rightarrow T\ a_1\ \ldots\ a_k \tag{14.1}$$

The expression "$T\ a_1\ \ldots\ a_k$" must be well-formed, in the sense that it must be well-typed and it must fulfill the constraints for parametric arguments. Moreover, the type T cannot appear among the arguments $a_1\ \ldots\ a_k$, even if the typing rules could allow it. For instance, the following definition is rejected:

```
Inductive T : Set→Set := c : (T (T nat)).
```
Error: Non strictly positive occurrence of "T" in "T (T nat)"

14.1.2.2 Positivity Constraints

The type description in 14.1 may lead the reader to believe that the constructor type has to be a non-dependent type. Of course not. Nevertheless, there is a constraint on the expressions $t_1, \ldots, t_l$. Each of these terms may be a constant or a function and the inductive type may only appear inside the final type of this function. If t_i is the type of a constant then t_i can have the form "$g\ (T\ b_{1,1}\ \ldots\ b_{1,k}) \cdots (T\ b_{l,1}\ \ldots\ b_{l,k})$," provided the expressions $b_{i,j}$ also satisfy the typing rules, and T does not occur in these expressions or in g. If t_i is the type of a function, this type can have the form

$$t'_1 \rightarrow \ldots \rightarrow t'_m \rightarrow g\ (T\ b_{1,1}\ \ldots\ b_{1,k}) \cdots (T\ b_{l,1}\ \ldots\ b_{l,k})$$

but the type T cannot occur in the expressions $t'_1\ \ldots\ t'_m$ or in the expressions $b_{i,j}$. These constraints are called the *strict positivity constraints*.

For instance, the following inductive definition is well-formed. It generalizes the notion of infinitely branching trees given in Sect. 6.3.5.2:

```
Inductive inf_branch_tree (A:Set) : Set :=
  inf_leaf : inf_branch_tree A
| inf_node : A →(nat → inf_branch_tree A)→ inf_branch_tree A.
```

The first constructor has a constant type and satisfies the conditions about using the inductive type in the final type and about parameter arguments. The type of second constructor expresses that there are two arguments and this type is well-formed from the point of view of the typing rules. The first argument of the second constructor does not involve the inductive type but the parameter; the second argument is a function, whose final type is the inductive type applied to the parameter. The argument type for this function is another type, so there is no problem.

On the other hand, the following definition is a simple example that violates the positivity rule:

```
Inductive T : Set := l : (T→T)→T.
```
Error: Non strictly positive occurrence of "T" in "(T→T)→T"

A more precise description of the formation rules for inductive types is given in the *Coq* system documentation and in the article [69], but we can already try to understand what is wrong with this definition. If this type `T` was accepted, we would also be allowed to define the following functions:

```
Definition t_delta : T :=
 (l fun t:T ⇒ match t with (l f) ⇒ f t end).

Definition t_omega: T :=
  match t_delta with l f ⇒ (f t_delta) end.
```

The expression `t_omega` could be reduced by ι-reduction to the following expression:

```
(fun t:T ⇒ match t with l f ⇒ f t end t_delta)
```

and after β-reduction:

```
match t_delta with l f ⇒ f t_delta end
```

We would have yet another expression that can be ι-reduced but this expression is the same as the initial expression and the process could go on for ever. Allowing this kind of inductive construction would mean losing the property that reductions always terminate and it would no longer be decidable to check whether a given term has a given type. This example shows that inductive types with non-strictly positive occurrences are a danger to the type-checking algorithm. Another example shows that they even endanger the consistency of the system, by making it possible to construct a proof of `False`. If the inductive type `T` were accepted, we would be able to define the following function:

```
Definition depth : T→nat :=
  fun t:T ⇒ match t with l f ⇒ S (depth (f t)) end.
```

With one ι-reduction we would obtain the following equality:

```
depth (l (fun t:T ⇒ t)) =
    S (depth ((fun t:T ⇒ t) (l (fun t:T ⇒ t))))
```

and with one more β-reduction we would obtain:

```
(depth (l (fun t:T ⇒ t)) = (S (depth (l (fun t:T ⇒ t))).
```

This is contradictory to the theorem `n_Sn`:

n_ Sn : ∀ *n:nat, n* ≠ *S n*

14.1.2.3 Universe Constraints

An inductive definition actually contains several terms that are types. One of these terms is the type of the type being defined the others are the type of the constructors. Each of these type terms also has a type, and the inductive definition is only well-formed if the type of all these type expressions is the same up to convertibility.

For instance, the type of natural numbers is declared as type `nat` of the sort `Set`. The first constructor is `O` of type `nat`, and `nat` has the type `Set`. The second constructor is `S` of type `nat→nat`, and `nat→nat` has the type `Set`, thanks to the first line of the table `triplets`. The type of `nat` and the types of the constructors coincide: we do have a well-formed definition.

This universe constraint plays a more active role when considering types where a constructor contains a universal quantification over a sort. These types are especially useful when we want to consider mathematical structures that are parameterized with respect to a carrier set, like the following definition of a group:

```
Record group : Type :=
  {A : Type;
   op : A→A→A;
   sym : A→A;
   e : A;
   e_neutral_left : ∀x:A, op e x = x;
   sym_op : ∀x:A, op (sym x) x = e;
   op_assoc : ∀x y z:A, op (op x y) z = op x (op y z);
   op_comm : ∀x y:A, op x y = op y x}.
```

This record type is an inductive type with a constructor named `Build_group`.

```
Check Build_group.
```
Build_ group : ∀ *(A:Type)(op:A→A→A)...*

The type of the carrier `A` is `Type`. As we have seen in Sect. 2.5.2, this type actually hides a type `Type(`i`)` for some i and the whole type for the constructor necessarily has the type `Type(`j`)` for some index j that has to be greater than

i. Therefore, the type of `group` actually hides an index that has to be higher than 0. Using the type convertibility described in Sect. 2.5.2 we can make sure that the type of the constructor's type is equal to the type of the inductive type.

We can change our definition of `group` to insist that the type of `A` should be `Set` (thus making our description less general), but we cannot specify that the type of the whole group structure should be `Set`, because the constructor type necessarily belongs to a universe that is higher in the hierarchy than `Set`.

In previous versions of *Coq*, the typing rules of Fig. 4.4 were designed in such a way that the sort `Set` was impredicative and the definition of a group structure in this sort was possible. Still, other restrictions had to be enforced to ensure the consistency of the system and there was a distinction between *strong* and *weak* elimination. Strong elimination was usable to obtain values in the type `Type`, while weak elimination was usable to obtain regular values (like numbers, boolean values, and so on). Strong elimination was not allowed on types of the sort `Set` that had constructors with a quantification over a sort (these were called large constructors). As a consequence, a group structure in the sort `Set` was still difficult to use because the access function that returned the carrier type could not be defined.

14.1.3 Building the Induction Principle

This section is reserved for the curious reader and can be overlooked for practical purposes. Induction principles are tools to prove that some properties hold for all the elements of a given inductive type. For this reason, all induction principles have a *header* containing universal quantifications, finishing with a quantification over some predicate P ranging over the elements of the inductive type. Then come a collection of implications whose premises we call the *principal premises*. At the end there is an *epilogue*, which always asserts that the predicate P holds for all the elements of the inductive type. We first describe the header and the epilogue, before coming back to the principal premises.

In the rest of this section, we consider that we are studying the induction principle for an inductive type T.

Generating the Header

If T is a dependent type (a function) it actually represents a family of inductive types indexed by the expressions that appear as arguments of T, among which the parameters play special a role.

Inductive definitions with parameters are built like inductive definitions without parameters that are constructed in a context where the parameters are fixed. The definition is later generalized to a context where the parameter is free to change. This step of generalizing the definition involves inserting universal quantification in the types of all the elements of the definition: the

type itself, the constructors, and the induction principle. For instance, the induction principle for polymorphic lists (see Sect. 6.4.1) and the induction principle for fixed-height trees (see Sect. 6.5.2) both start with a universal quantification over an element `A` of the sort `Set`:

```
∀A:Set, ...
```

For the arguments of T that are not parameters, we have to make sure that the predicate we want to prove over all elements can follow the variations of these arguments. We need to express explicitly that this predicate can depend on these arguments. Thus, it receives not one argument but $k + 1$, where k is the number of non-parametric arguments of the inductive T. For instance, the type of natural numbers is a constant type, therefore the predicate is a one-argument predicate, the element of type `nat` that is supposed to satisfy the property. Here is the header, with no quantification over parameters, and quantification over a oneargument predicate:

```
∀P:nat→Prop, ...
```

For the induction principle over polymorphic lists, there is one parameter, but the type does not depend on other arguments. Here is the header, with one quantification over a parameter and another quantification over a one argument predicate:

```
∀(A:Set)(P:list A → Prop), ...
```

For the induction principle over fixed-height trees, there is one parameter and the type depends on an extra argument, which is an integer. Here is the header, with one universal quantification over a parameter and another quantification over a two-argument predicate, an integer `n` and a tree of type "`htree A n`":

```
∀(A:Set)(P:∀n:nat, htree A n → Prop), ...
```

As we see, when the type is dependent, the predicate over which the header quantifies also has a dependent type.

Generating the Epilogue

After the header and the principal premises comes the conclusion of the induction principle. This conclusion simply states that the predicate is satisfied by all elements of the inductive type, actually all elements of all members of the indexed family of types. There is no quantification over the parametric arguments, because the whole induction principle is in the scope of such a quantification, but there is a quantification over all possible values of the dependent arguments, then a quantification over all elements of the indexed type, and the predicate is applied to all the arguments of the dependent type and this element. For instance, the induction principle for polymorphic lists has the following epilogue:

```
...∀l:list A, P l.
```

For fixed-height trees, the epilogue takes the following shape, due to the presence of a non-parametric argument:

```
...∀n:nat, ∀t:htree A n, P n t
```

Generating the Principal Premises

Intuitively, the principal premises are given to make sure that the predicate has been verified for all possible uses of the constructors. Moreover, the inductive part of the principle is that we can suppose that the predicate already holds for the subterms in the inductive type when proving that it holds for a whole term. Thus, the principal premises will contain a universal quantification for all possible arguments of the constructor with an extra induction hypothesis for every argument whose type is the inductive type T.

When one of the arguments is a function, as in the type `Z_fbtree` of Sect. 6.3.5.1, it is enough to quantify over such a function, but when the function's final type is the inductive type T, the subterms to consider for the induction hypotheses are all the possible images of this function.

The principal premise finishes with an application of the predicate P to the constructor being considered, itself applied to all the arguments, first the parameters of the inductive definition, then the arguments that are universally quantified in the premise, excluding the induction hypotheses. For instance, the principal premise for the `January` constructor of the type `month` (see Sect. 6.1.1) does not contain any quantification, because the constructor is a constant:

```
P January
```

For the `nil` constructor of the `list` type, the parameter naturally appears, but there is no quantification:

```
P nil
```

For the `bicycle` constructor of the `vehicle` type (see Sect. 6.1.6), the only argument is not in the inductive type and there is only one universal quantification:

```
∀n:nat, P (bicycle n)
```

For the `S` constructor of the `nat` type, the only argument is in the inductive type, so there is one universal quantification over `n:nat` and one induction hypothesis "`P n`":

```
∀n:nat, P n → P (S n)
```

For the `Z_bnode` constructor of type `Z_btree` (see Sect. 6.3.4), the first argument is not in the inductive type, but the other two are. The principal premise has three universal quantifications and two induction hypotheses:

```
∀z:Z, ∀t0:Z_btree, P t0 →
      ∀t1:Z_btree, P t1 → P (Z_bnode z t0 t1)
```

For the `Z_fnode` constructor of the `Z_fbtree` type, the first argument is not in the inductive type and the second one is a function whose final type is in the inductive type. There are two universal quantifications over these arguments and one induction hypothesis that is universally quantified over all possible arguments of the constructor's function argument:

```
∀(z:Z)(f:bool→Z_fbtree), (∀x:bool, P (f x))→ P (Z_fnode z f)
```

When considering a dependent inductive type, induction hypotheses must be adapted to the right arguments for the whole premise to be well-typed. For instance, the `hnode` constructor of the `htree` type (see Sect. 6.5.2) takes five arguments: the first one is a parameter, the second and third ones are not in the inductive type, and the last two are, with a different value for a dependent argument. The principal premise contains universal quantifications for the four elements that are not parameters; for the two arguments in the inductive type, the dependent argument is adjusted as in the constructor definition and the same adjustment is done for the predicate P in the induction hypotheses:

```
∀(n:nat)(x:A)(t:htree A n), P n t →
   ∀t':htree A n, P n t' → P (S n)(hnode A n x t t')
```

The header, the principal premises, and the epilogue are grouped together to obtain the type of the induction principles.

14.1.4 Typing Recursors

Recursively defined functions over an inductive type may have a dependent type. For each inductive type of the sort `Set`, the *Coq* system actually generates a recursive function with a dependent type, whose name is obtained by concatenating the suffix `_rec` to the type name. For instance, the function `nat_rec` is provided for the type of natural numbers:

```
nat_rec:∀P:nat→Set, P 0 →(∀n:nat, P n → P (S n))→
   ∀n:nat, P n
```

This function may be used to define recursive functions instead of the `Fixpoint` command or the `fix` construct. We call this function the *recursor* associated with an inductive type. The curious reader may wonder what the relation is between this recursor and the `Fixpoint` command. The answer is that the recursor is defined using the `Fixpoint` command, but we will show progressively how this recursor is an evolution from simple notions of recursors. We will then show the relation between the recursor and the induction principle of an inductive type.

Non-dependent Recursion

Most often, recursive functions over natural numbers have the following shape:

```
Fixpoint f (x:nat) : A :=
 match x with
   O ⇒ exp1
 | S p ⇒ exp2
 end.
```

We have willingly left A, exp_1, and exp_2 unknown in this declaration, to indicate that these are the elements that change from one function to another. Nevertheless, exp_1 and exp_2 both have to be well-typed and of type A. For exp_2 the situation is slightly more complicated, because the pattern "`S p ⇒...`" is a binding construct and the variable `p` may be used inside exp_2. Moreover, the structural recursion conditions also make it possible to use the value (`f p`) inside exp_2.[1] All this can be expressed by saying that the previous declaration scheme is practically equivalent to the following one:

```
Fixpoint f (x:nat) : A :=
 match x with
   O ⇒ exp1
 | S p ⇒ exp2' p (f p)
end.
```

Here, A, exp_1, and exp_2' must be closed expressions where f cannot occur and they must have the following types:

$$A : \texttt{Set} \qquad exp_1 : A \qquad exp_2' : nat \to A \to A$$

In practice, most simple recursive functions are defined by A, exp_1, and exp_2', so that they could actually be described using a function `nat_simple_rec` that has the following dependent type:

```
nat_simple_rec:∀ A:Set, A →(nat→A→A)→nat→A
```

Actually, this function `nat_simple_rec` can be defined in *Coq* using the following command:

```
Fixpoint nat_simple_rec (A:Set)(exp1:A)(exp2:nat→A→A)(x:nat)
 {struct x} : A :=
  match x with
  | O ⇒ exp1
  | S p ⇒ exp2 p (nat_simple_rec A exp1 exp2 p)
  end.
```

The set of functions that can be described using `nat_simple_rec` contains all primitive recursive functions in the sense of Dedekind [34]. But because `A` may also be a function type, this set also contains functions that are not primitive

[1] It is actually mandatory that at least one recursive call appears in the whole function definition.

recursive, as was announced by Hilbert [49] and shown by Ackermann [1] (see Sect. 4.3.3.2). From the point of view of the Calculus of Inductive Constructions, we cannot use the function `nat_simple_rec` to define all interesting functions, because the dependently typed functions are missing.

Dependent Pattern Matching

For dependently typed functions, the target type is not given by a simple element of the `Set` sort, but by a function associating a type with every element of the domain.

If we want to build a dependently typed function, the pattern matching construct naturally plays a role, because we can associate different computations to different values only by using this construct. We are led to build pattern matching constructs where the expressions given in each case have a different type. This makes type synthesis too difficult for the automatic typechecker and we have to help it by providing the function that describes the type variation. For instance, we can consider the type of boolean values and define a function that maps `true` to `0` (of type `nat`) and `false` to `true` (of type `bool`). We first have to describe the function that maps each element of `bool` to the output type:

```
Definition example_codomain (b:bool) : Set :=
  match b with true ⇒ nat | false ⇒ bool end.
```

The second step is to build a dependent pattern matching construct that uses this function to control how the various branches are typed:

```
Definition example_dep_function (b:bool) : example_codomain b :=
  match b as x return example_codomain x with
  | true ⇒ 0
  | false ⇒ true
  end.
```

The user must provide a typing indication, giving a variable after the keyword "`as`" and an expression after the keyword "`return`," which may depend on the variable. The type of this variable is the type of the expression that is the object of the pattern matching construct.

If a pattern matching rule has the form

$$pattern \Rightarrow exp$$

and the typing indication has the form "`as` x `return` t," then the expression exp must have the type $t\{x/pattern\}$.

An alternative approach to explaining dependent pattern matching is to associate a function F with the fragment "`as` x `return` t," with the following value:

$$\texttt{fun}\ x \Rightarrow \texttt{t}$$

From this point of view, we say that the expression *exp* in the pattern matching rule must have the type "*F pattern*."

When the expression that is the object of the pattern matching construct has a dependent type, the function F cannot be a function with only one argument, because the dependent arguments for the type are required. The user must then indicate in which instance of the dependent type each pattern is considered, using the following form, where the parametric arguments of the inductive type must be replaced by jokers "`_`":

`as` *x* `in` *type_name* `_ _` *c d* `return` *t*

When the filtered expression has the type "*type_name A B c d*," the function F has the following shape:

`fun` *c d* ⇒ `fun x:`*type_name A B c d* ⇒ *t*

As we did in the previous section for simple recursion, we can represent the dependent pattern matching simply with a dependently typed function `bool_case` that takes as argument the function `F:bool→Set` and the two expressions of type "`F true`" and "`F false`." This function `bool_case` has the following type:

```
bool_case:∀F:bool→Set, F true → F false → ∀x:bool, F x
```

It can actually be defined in *Coq* with the following command:

```
Definition bool_case
   (F:bool→Set)(v1:F true)(v2:F false)(x:bool) :=
  match x return F x with true ⇒ v1 | false ⇒ v2 end.
```

The same kind of function can be constructed for case-by-case computation on natural numbers. The function has the following type:

```
nat_case:∀F:nat→Set, F 0 →(∀m:nat, F (S m))→∀n:nat, F n
```

This function can be defined with the following command:

```
Definition nat_case
  (F:nat→Set)(exp1:F 0)(exp2:∀p:nat, F (S p))(n:nat) :=
  match n as x return F x with
  | O ⇒ exp1
  | S p ⇒ exp2 p
  end.
```

Dependently Typed Recursors

We can now combine dependent pattern matching and recursion to find the form of the most general recursor associated with an inductive type. The first argument of this recursor must be a function f mapping elements of the inductive type to elements of the sort `Set`.

Then, for every constructor, we have to provide an expression whose type is built from the arguments of this constructor. If the constructor is c and the arguments are $a_1 : t_1, \ldots, a_k : t_k$ then the expression can use the values a_i and the values corresponding to recursive calls on those a_i that have the inductive type. The expression associated with a constructor must then have the type "$\forall (b_1 : t'_1) \cdots (b_l : t'_l),\ c\ b_{i_1} \ldots b_{i_k}$" where l is k plus the number of indices i such that t_i contains an instance of the inductive type being studied. Each of the arguments a_i is associated with the arguments b_{j_i} in the following manner:

- $j_1 = 1$ and $t'_1 = t_1$.
- If t_i does not contain an instance of the inductive type being studied, then $j_{i+1} = j_i + 1$ and $t'_{j_i} = t_i$.
- If t_i is an instance of the inductive type being studied, then $j_{i+1} = j_i + 2$, $t'_{j_i} = t_i$, and $t_{j_i+1} = (f\ b_{j_i})$.
- If t_i is a function type $\forall (c_1 : \tau_1) \cdots (c_m : \tau_m),\ \tau$ where τ is an instance of the inductive type, then $j_{i+1} = j_i + 2$, $t_{j_i} = t_i$, and

$$t_{j_i+1} = \forall (c_1 : \tau_1) \cdots (c_m : \tau_m),\ f\ (b_{j_i}\ c_1 \cdots c_m)$$

To illustrate this process, we study how the recursor for the type of the natural numbers is built. This recursor starts by taking an argument of type f : `nat→Set`. Then, there are two values. The first one must have type "f `0`," since "`0`" is a constant. For the second constructor there is only one argument, say a_1 of the type `nat`, and therefore we have an argument b_1 of the type `nat` and an argument b_2 of the type "$f\ b_1$." The whole expression has the following type:

$\forall$`(`b_1`:nat)(`b_2`:(f` b_1`)), f (S` b_1`).`

With a different choice of bound variable names and taking into account the non-dependent products, this can also be written

```
∀n:nat, f n → f (S n).
```

Putting together all the elements of this recursor, we get the following type:

```
nat_rec :
  ∀f:nat→Set, f 0 →(∀n:nat, f n → f (S n))→ ∀n:nat, f n.
```

This recursor is automatically built when the inductive type is defined. This construction is equivalent to the following definition:

```
Fixpoint nat_rec (f:nat→Set)(exp1:f 0)
 (exp2:∀p:nat, f p → f (S p))(n:nat){struct n} : f n :=
  match n as x return f x with
  | 0 ⇒ exp1
  | S p ⇒ exp2 p (nat_rec f exp1 exp2 p)
  end.
```

We see that this function basically has the same structure as the simple recursor described at the start of this section. It can be used instead of the `Fixpoint` command to define most recursive functions. For instance, the function that multiplies a natural number by 2 can be described in the following manner:

```
Definition mult2' :=
  nat_rec (fun n:nat ⇒ nat) 0 (fun p v:nat ⇒ S (S v)).
```

On the other hand, it is difficult to define multiple step recursive functions, like the function `div2` (see Sect. 9.3.1).

The functions `nat_rec` and `nat_ind` practically have the same type, except that `Set` is replaced by `Prop` in the recursor type. This is another instance of the Curry–Howard isomorphism between programs and proofs. The induction principle is actually constructed in exactly the same way as the recursor and is a function that constructs a proof of "$P\ n$" by a recursive computation on n. This function could have been described in the following manner:

```
Fixpoint nat_ind (P:nat→Prop)(exp1:P 0)
 (exp2:∀p:nat, P p → P (S p))(n:nat){struct n} : P n :=
  match n as x return P x with
  | 0 ⇒ exp1
  | S p ⇒ exp2 p (nat_ind P exp1 exp2 p)
  end.
```

As a second example, we can study how the recursor for binary trees with integer labels is constructed (see Sect. 6.3.4). Here again, the recursor takes as first argument a function of the type `f_btree→Set`, then the expressions for each constructor. For the first constructor, a constant, the value must have the type "`f Z_leaf`." For the second constructor, there are three arguments, which we can name a_1 : `Z`, a_2 : `Z_btree`, a_3 : `Z_btree`. We work progressively on these arguments in the following manner:

1. $j_1 = 1$ and b_1 must have the type `Z`.
2. $j_2 = 2$ and b_2 must have the type `Z_btree`.
3. Since `Z_btree` is the inductive type being studied, $j_3 = 4$ and b_3 must have the type $(f\ b_2)$.
4. b_4 must have type `Z_btree`.
5. Since `Z_btree` is the inductive type being studied, there must be an argument b_5 with the type $(f\ b_4)$.

The type of the expression for this constructor has the following shape:

```
∀(b1:Z)(b2:Z_btree)(b3:f b2)
    (b4:Z_btree)(b5:f b4), f (Z_bnode b1 b2 b4)
```

With a different choice of bound variable names and taking into account the non-dependent products, this can also be written

```
∀(z:Z)(t1:Z_btree), f t1 →
    ∀t2:Z_btree, f t2 → f (Z_bnode z t1 t2)
```

Putting together all the elements of this recursor we get the following type:

```
Z_btree_rec :
  ∀f:Z_btree→Set,
   f Z_leaf →
   (∀(n:Z)(t1:Z_btree), f t1 →
      ∀t2:Z_btree, f t2 → f (Z_bnode n t1 t2))→
   ∀t:Z_btree, f t.
```

The definition of a recursor with this type is obtained with the following command:

```
Fixpoint Z_btree_rec (f:Z_btree→Set)(exp1:f Z_leaf)
 (exp2:∀(n:Z)(t1:Z_btree), f t1 →
       ∀t2:Z_btree, f t2 → f (Z_bnode n t1 t2))
 (t:Z_btree){struct t} : f t :=
  match t as x return f x with
  | Z_leaf ⇒ exp1
  | Z_bnode n t1 t2 ⇒
      exp2 n t1 (Z_btree_rec f exp1 exp2 t1) t2
        (Z_btree_rec f exp1 exp2 t2)
  end.
```

Here again, this recursor has practically the same type as the induction principle and we can see the induction principle as a function to build proofs about trees using a recursive computation on these trees.

As a third example, we consider the type of functional binary trees described in Sect. 6.3.5.1. Here is the second constructor:

```
Z_fnode:Z→(bool→Z_fbtree)→Z_fbtree
```

We know that $t_1 =$ `Z` and $t_2 =$ `bool→Z_fbtree`. Here are the steps to construct the expression associated with this constructor:

1. $j_1 = 1$ and $t'_1 =$ `Z`.
2. $j_2 = 2$ and $t'_2 =$ `bool→Z_fbtree`.
3. Since t_2 has `Z_fbtree` as head type there must be a type

$$t'_3 = \forall\, c_1\texttt{:bool}, f\ (b_2\ c_1)$$

The type of the expression for this constructor has the following shape:

```
∀(b1:Z)(b2:bool→Z_fbtree)(b3:∀c1:bool, f (b2 c1)),
       f (Z_fnode b1 b2)
```

With a different choice of bound variable names and taking into account the non-dependent products, this can also be written

```
∀(z:Z)(g:bool→Z_fbtree), (∀b:bool, f (g b))→ f (Z_fnode z g)
```

We still have not described how the recursors are built for inductive dependent types. The curious reader should refer to Paulin-Mohring's work on this matter [69, 70].

Exercise 14.1 *Describe natural number addition using* `nat_rec` *instead of the command* `Fixpoint`.

Exercise 14.2 ** Redefine the induction principle* `Z_btree_ind` *using the* `fix` *construct.*

14.1.5 Induction Principles for Predicates

The induction principle that is generated by default for inductive predicates (inductive types of the `Prop` sort) is different from the induction principle that is generated for the "twin" inductive type that has the same definitions and constructors but is in the `Set` sort. Actually, the induction principle for types of the sort `Prop` is simplified to express proof irrelevance.

In this section, we describe the differences between *maximal induction principles* and *simplified induction principles*. The maximal induction principle is the one obtained through the technique given in Sect. 14.1.3. The simplified induction principle is the one that is constructed by default for types of the sort `Prop`.

Inductive types of the sort `Prop` usually are dependent. When considering an inductive type with n arguments, the maximal induction principle contains a universal quantification over a predicate with $n+1$ arguments, the argument of rank $n+1$ is in the type being considered, and the n other arguments are necessary to make a well-typed term. For the minimal induction principle, a predicate with only n arguments is used. The argument of rank $n+1$ is simply dropped.

The minimal induction principle is obtained by instantiating the maximal principle for a predicate with $n+1$ arguments that forgets the last one and refers to another predicate with n arguments.

For instance, the maximal induction principle for the inductive predicate `le` should have the following type:

```
∀ (n:nat)(P:∀ n0:nat, n ≤ n0 → Prop),
   P n (le_ n n)→(∀ (m:nat)(l:n ≤ m), P m l → P (S m)(le_ S n m l))→
   ∀ (n0:nat)(l:n ≤ n0), P n0 l
```

Instantiating this type for the predicate "`fun (m:nat) (_:n≤m) ⇒ (P m)`" gives the type that we are accustomed to see for `le`. We can check this with the help of the *Coq* system:

```
Eval compute in
  (∀ (n:nat)(P:nat→Prop),
     (fun (n':nat)(P:∀m:nat, n' ≤ m → Prop) ⇒
```

```
      P n' (le_n n')→
      (∀(m:nat)(h:n' ≤ m), P m h → P (S m)(le_S n' m h))→
      ∀(m:nat)(h:n' ≤ m), P m h) n
     (fun (m:nat)(_:n ≤ m) ⇒ P m)).
```

= ∀ (n:nat)(P:nat→Prop),
P n →(∀ m:nat, n ≤ m → P m → P (S m))→
∀ m:nat, n ≤ m → P m
: Prop

The maximal induction principle can always be obtained using the `Fixpoint` command. Here is an example for the predicate `even` (this predicate is defined in Sect. 8.1). This proof is a structurally recursive function whose principal argument is a proof that some number is even.

```
Fixpoint even_ind_max (P:∀n:nat, even n → Prop)
 (exp1:P 0 O_even)
 (exp2:∀(n:nat)(t:even n),
        P n t → P (S (S n))(plus_2_even n t))(n:nat)(t:even n)
 {struct t} : P n t :=
  match t as x0 in (even x) return P x x0 with
  | O_even ⇒ exp1
  | plus_2_even p t' ⇒
    exp2 p t' (even_ind_max P exp1 exp2 p t')
  end.
```

The simplified induction principle can also be obtained with a `Fixpoint` command. Here again, the induction principle is a function whose principal argument is the proof of an inductive predicate:

```
Fixpoint even_ind' (P:nat→Prop)(exp1:P 0)
 (exp2:∀n:nat, even n → P n → P (S (S n)))(n:nat)(t:even n)
 {struct t} : P n :=
  match t in (even x) return P x with
  | O_even ⇒ exp1
  | plus_2_even p t' ⇒ exp2 p t' (even_ind' P exp1 exp2 p t')
  end.
```

As we see in this definition, the typing information added to the dependent pattern matching construct does contain an `as` part. This corresponds to the fact that `P` has only one argument instead of two. The simplified induction principle for the predicate `clos_trans` on relations (see Sect. 8.1.1) can also be reconstructed with the following definition:

```
Fixpoint clos_trans_ind' (A:Set)(R P:A→A→Prop)
 (exp1:∀x y:A, R x y → P x y)
 (exp2:∀x y z:A,
        clos_trans A R x y →
        P x y → clos_trans A R y z → P y z → P x z)(x y:A)
```

```
(p:clos_trans A R x y){struct p} : P x y :=
match p in (clos_trans _ _ x x0) return P x x0 with
| t_step x' y' h ⇒ exp1 x' y' h
| t_trans x' y' z' h1 h2 ⇒
  exp2 x' y' z' h1 (clos_trans_ind' A R P exp1 exp2 x' y' h1)
       h2 (clos_trans_ind' A R P exp1 exp2 y' z' h2)
 end.
```

Exercise 14.3 *** Manually build the induction principle for the* `sorted` *predicate from Sect. 8.1.*

14.1.6 The `Scheme` Command

In the *Coq* system, the simplified induction principle is automatically generated for inductive types of the sort `Prop` and the maximal induction principle is automatically generated for the other inductive type definitions. To obtain a maximal induction principle for an inductive type in the `Prop` sort, it is possible to perform a manual construction as we did for `even_ind_max`, but it is also possible to use a special command, called `Scheme`. Here is an example of use:

```
Scheme even_ind_max := Induction for even Sort Prop.
```

The keyword `Induction` means that a maximal induction principle is requested. This keyword can be replaced with the keyword `Minimality` to obtain a simplified induction principle.

We can also produce simple recursors with the `Scheme` command. For instance, the simple recursor `nat_simple_rec` described in Sect. 14.1.4 can be obtained with the following command:

```
Scheme nat_simple_rec := Minimality for nat Sort Set.
```

Exercise 14.4 ** Redo the proof of the theorem* `le_2_n_pred` *from Sect. 9.2.4 using the maximal induction principle for* `le`*.*

14.2 *** Pattern Matching and Recursion on Proofs

With one notable exception, the formation rules for pattern matching preclude that a term of the sort `Type` or `Set` can be obtained through a pattern matching construct of expressions of the sort `Prop`. This restriction ensures proof irrelevance. This is necessary to ensure that the extraction process is a safe way to produce code.

14.2.1 Restrictions on Pattern Matching

When building a function of the type "$\forall x$:A, `P x` $\rightarrow$ `T x`" where "`P x`" has sort `Prop` and "`T x`" has the sort `Set`, the restriction makes it difficult to build a function that is recursive over the proof of "`P x`." A common solution is to perform a proof by induction on x, to determine the value in each case, and to use inversions on "`P x`" to obtain the needed arguments for recursive calls. For instance, consider a strongly specified function for subtracting natural numbers:

```
Definition rich_minus (n m:nat) := {x : nat | x+m = n}.

Definition le_rich_minus : ∀m n:nat, n ≤ m → rich_minus m n.
```

This definition cannot be obtained using a direct elimination of the hypothesis `n`$\leq$`m`, but we can use a proof by induction over arguments of type `nat`.

```
induction m.
intros n Hle; exists 0.
```

```
...
Hle : n ≤ 0
============================
 0+n = 0
```

This goal statement is in sort `Prop` and pattern matching on hypothesis `Hle` is not restricted. Pattern matching actually occurs in the `inversion` tactic. It returns a single trivial goal:

```
inversion Hle; trivial.
```

The proof continues with the step case:

```
intros n; case n.
intros Hle; exists (S m).
```

```
...
IHm : ∀ n:nat, n ≤ m → rich_minus m n
n : nat
Hle : 0 ≤ S m
============================
 S m + 0 = S m
```

Here again, the goal statement is in the sort `Prop` and we can build a proof without any restriction.

```
auto with arith.
intros n' Hle.
```

```
...
IHm : ∀ n:nat, n ≤ m → rich_minus m n
n : nat
n' : nat
```

Hle : S n' ≤ S m
==========
rich_minus (S m)(S n')

At this point, the goal statement and the hypothesis `IHm` are both in the sort `Set` and we can reason by cases on the instantiation of the hypothesis for `n'`.

```
elim (IHm n').
intros r Heq.
exists r.
rewrite <- Heq; auto with arith.
```

Now, the goal statement is in the sort `Prop` and pattern matching on the hypothesis `Hle` is possible. We can use the `inversion` tactic and conclude:

```
inversion Hle; auto with arith.
Defined.
```

Exercise 14.5 *** We consider the inductive property on polymorphic lists "u is a prefix of v":*

```
Set Implicit Arguments.

Inductive lfactor (A:Set) : list A → list A → Prop :=
  lf1 : ∀u:list A, lfactor nil u
| lf2 : ∀(a:A)(u v:list A),
       lfactor u v → lfactor (cons a u)(cons a v).
```

Build a function realizing the following specification:

```
∀(A:Set)(u v:list A), lfactor u v → {w : list A | v = app u w}
```

14.2.2 Relaxing the Restrictions

The main exception to the rule that pattern matching cannot be done on proofs when aiming for data of the type `Type` or the sort `Set` occurs when the inductive type has only one constructor and this constructor only takes arguments whose type has type `Prop`. Intuitively, pattern matching on this kind of property is acceptable because no data of sort `Type` or `Set` can be obtained in this manner. When a definition is parametric, only the regular arguments are constrained to inhabit a type of sort `Prop`. The relaxed condition for pattern matching on an inductive type of sort `Prop` is used extensively for equality. It is remarkable that this kind of pattern matching directly provides the possibility of representing rewriting. Equality is described by the following inductive definition:

```
Inductive eq (A:Type)(x:A) : A→Prop :=
    refl_equal : eq A x x.
```

If we forget about the parametric arguments, the constructor `refl_equal` is a constant and pattern matching is allowed for all elements of this type to obtain elements of type `Type` and `Set`. The constants provided in the *Coq* libraries could have been obtained using the following definitions:

```
Definition eq_rect (A:Type)(x:A)(P:A→Type)(f:P x)(y:A)(e:x = y)
  : P y := match e in (_ = x) return P x with
           | refl_equal ⇒ f
           end.

Definition eq_rec (A:Type)(x:A)(P:A→Set) :
  P x → ∀y:A, x = y → P y := eq_rect A x P.

Implicit Arguments eq_rec [A].
```

Here the definition of `eq_rec` relies on the convertibility between `Set` and `Type` presented in Sect. 2.5.2.

The function `eq_rec` is mainly useful for dependent types. If we need to construct data of the type "`P x`," we know that `x=y` holds, and we have a value `a` of the type "`P y`," then the value `a` can be returned after we have shown that its type can be rewritten in "`P x`." For instance, let us consider a type `A`, a function `A_eq_dec` that decides the equality of two expressions, a dependent type `B:A→Set`, a function "`f:∀x:A,B x`," and two values `a` of the type `A` and `v` of the type "`B a`." We want to define the function that coincides with `f` everywhere but in `a`, where the result is `v`. This function can be defined in the following manner:

```
Section update_def.
  Variables (A : Set)(A_eq_dec : ∀x y:A, {x = y}+{x ≠ y}).
  Variables (B : A→Set)(a : A)(v : B a)(f : ∀x:A, B x).

  Definition update (x:A) : B x :=
    match A_eq_dec a x with
    | left h ⇒ eq_rec a B v x h
    | right h' ⇒ f x
    end.
End update_def.
```

Reasoning about the function `eq_rec` is difficult because we reach the limits of the expressive power of inductive types. To solve this difficulty, the *Coq* system provides an extra axiom `eq_rec_eq` (in the module `Eqdep`) that expresses the intuitive meaning of `eq_rec`. Its result is the same as its input.

```
eq_rec_eq
     :∀(U:Type)(Q:U→Set)(p:U)(x:Q p)(h:p = p),
                 x = eq_rec p Q x p h
```

Exercise 14.6 *** Show that the* `update` *function satisfies the following proposition:*

```
update_eq
     :∀(A:Set)(eq_dec:∀x y:A, {x = y}+{x ≠ y})
                 (B:A→Set)(a:A)(v:B a)(f:∀x:A, B x),
                 update A eq_dec B a v f a = v.
```

14.2.3 Recursion

In spite of the restriction on pattern matching, it is possible to define recursive functions with a result in the `Set` sort and a principal argument in the `Prop` sort. The principal argument can then only be used to ensure the termination of the algorithm, but not to control the choices that decide which value is returned.

The best known example of recursion over an inductive predicate for a result in the `Set` sort is provided by well-founded induction that relies on an inductive notion of accessibility or adjoint [2, 52]. This notion of accessibility is described by the following inductive definition:

```
Inductive Acc (A:Set)(R:A→A→Prop) : A→Prop :=
  Acc_intro : ∀x:A, (∀y:A, R y x → Acc R y)→ Acc R x.
```

If R is an arbitrary relation, we say that a sequence $a_i (i \in \mathbb{N})$ is R-decreasing if "$R\ a_{i+1}\ a_i$" holds for every index i. If Φ is the predicate "does not belong to an infinite R-decreasing sequence," the following property holds:

$$\forall x.(\forall y.R\ y\ x \rightarrow \Phi\ y) \leftrightarrow \Phi\ x$$

Intuitively, if x belonged to an infinite R-decreasing sequence, the successor of y in that sequence would also belong to an infinite R-decreasing chain. In this sense, the accessibility predicates gives a good constructive description of the elements that do not belong to infinite decreasing chains. We can use this to express that function computations do not involve infinite sequences of recursive calls.

When h_x is a proof that some element x is accessible and y is the predecessor of x for the relation R, as expressed by a proof h_r of the type "$R\ y\ x$", we can easily build a proof that y is accessible by pattern matching. This new proof is *structurally smaller* than h_x. The theorem `Acc_inv` given in the *Coq* library performs this proof:

```
Print Acc_inv.
Acc_inv =
fun (A:Set)(R:A→A→Prop)(x:A)(H:Acc R x) ⇒
  match H in (Acc _ a) return (∀ y:A, R y a → Acc R y) with
  | Acc_intro x0 H0 ⇒ H0
  end
```

```
: ∀ (A:Set)(R:A→A→Prop)(x:A),
    Acc R x → ∀ y:A, R y x → Acc R y

Arguments A, R, x are implicit
Argument scopes are [type_scope _ _ _ _ _]
```

The pattern matching construct found in this definition is not restricted because the result is in the `Prop` sort.

The proof "`Acc_inv A R x H y Hr`" is structurally smaller than `H`, so it can be used as an argument in a recursive call for a function whose principal argument is `H`, even though the result is the sort `Set`. This is used in the following definition:

```
Fixpoint Acc_iter (A:Set)(R:A→A→Prop)(P:A→Set)
 (f:∀x:A, (∀y:A, R y x → P y)→ P x)(x:A)(H:Acc R x)
 {struct H} : P x :=
  f x (fun (y:A)(Hr:R y x) ⇒ Acc_iter P f (Acc_inv H y Hr)).
```

This function contains a recursive call, but apparently no pattern matching construct. Actually, the pattern matching construct is inside the `Acc_inv` function that is used only to provide the principal argument for the recursive call. This works only because `Acc_inv` is defined in a `transparent` manner and *Coq* is able to check that the recursive call follows the constraints of structural recursion. We obtain a function that is recursive over a proposition of the `Prop` sort and that outputs regular data whose type is in the `Set` sort.

This is used to define *well-founded recursive* functions. A relation on a type A is called *noetherian* if all elements of A are accessible. Every noetherian relation is also *well-founded*, in the sense that there is no element of A belonging to an infinite decreasing chain. By abusive notation, the *Coq* libraries use the term `well_founded` to denote noetherian relations. In classical logic the two notions are equivalent (see Exercise 15.7).

```
Print well_founded.
well_founded =
fun (A:Set)(R:A→A→Prop) ⇒ ∀ a:A, Acc R a
    : ∀ A:Set, (A→A→Prop)→Prop
Argument A is implicit
Argument scopes are [type_scope _]
```

When a relation is well-founded, we can define recursive functions where the relation is used to control which recursive calls are correct. This is expressed with a function `well_founded_induction` that is defined approximately as follows:

```
Definition well_founded_induction (A:Set)(R:A→A→Prop)
  (H:well_founded R)(P:A→Set)
  (f:∀x:A, (∀y:A, R y x → P y)→ P x)(x:A) : P x :=
  Acc_iter P f (H x).
```

The definitions of `Acc_iter` and `well_founded_induction` given in the *Coq* library may be different, but the ones we give here are correct and have the same type. The use of `well_founded_induction` is described in more detail in Sect. 15.2.

The scheme used in well-founded recursion to define a recursive function over a proposition to obtain a result in the sort `Set` can be reproduced with other inductive predicates, as long as we take care to isolate the pattern matching constructs in subterms in the `Prop` sort, as we did here for the transparent theorem `Acc_inv`. We give a detailed example in Sect. 15.4.

14.3 Mutually Inductive Types

It is possible to define *mutual* inductive types, where at least two inductive types refer to each other.

14.3.1 Trees and Forests

A typical example of a mutual inductive type provides trees where nodes can have an arbitrary but always finite number of branches. This can be expressed with a type where each node carries a list of trees, which we call a forest.

```
Inductive ntree (A:Set) : Set :=
  nnode : A → nforest A → ntree A
with nforest (A:Set) : Set :=
  nnil : nforest A | ncons : ntree A → nforest A → nforest A.
```

In this definition the type `ntree` uses the type `nforest` and the type `nforest` uses the type `ntree`. In this sense, the two types are mutually inductive.

To compute and reason on these mutually inductive types, the *Coq* system also provides a means to construct *mutually structural recursive functions*. For instance, the function that counts the number of nodes in a tree can be written in the following manner:

```
Open Scope Z_scope.

Fixpoint count (A:Set)(t:ntree A){struct t} : Z :=
  match t with
  | nnode a l ⇒ 1 + count_list A l
  end
 with count_list (A:Set)(l:nforest A){struct l} : Z :=
  match l with
  | nnil ⇒ 0
  | ncons t tl ⇒ count A t + count_list A tl
  end.
```

An unfortunate characteristic of mutual inductive types is that the *Coq* system generates induction principles that do not cover the mutual structure of these types:

```
ntree_ind :
  ∀(A:Set)(P:ntree A → Prop),
    (∀(a:A)(l:nforest A), P (nnode A a l))→
    ∀t:ntree A, P t.
```

This induction principle does not take into account the fact that the list that appears as a component of the `nnode` terms can contain subterms of the same type. As a result, this induction principle is practically useless. The induction principle that is associated with the type `nforest` is also excessively simplified and often unpractical.

Better induction principles can be obtained by using the `Scheme` command (already introduced in Sect. 14.1.6):

```
Scheme ntree_ind2 :=
   Induction for ntree Sort Prop
 with nforest_ind2 :=
   Induction for nforest Sort Prop.
```

This command actually generates two induction principles that are named as indicated in the command (here `ntree_ind2` and `nforest_ind2`). We can have a closer look at the first one:

```
ntree_ind2
    : ∀(A:Set)(P:ntree A → Prop)(P0:nforest A → Prop),
       (∀(a:A)(n:nforest A), P0 n → P (nnode A a n))→
       P0 (nnil A)→
       (∀n:ntree A,
          P n → ∀n0:nforest A, P0 n0 → P0 (ncons A n n0))→
       ∀n:ntree A, P n
```

This induction principle quantifies over two predicates: `P` is used for the type `ntree` and `P0` is used for the type `nforest`. There are three principal premises corresponding to the three constructors of the two types. Finally, the epilogue expresses that the predicate `P` holds for all trees of the type `ntree`. The induction principle for the type `nforest` has the same shape, only the epilogue differs. We present later a proof using this induction principle.

Mutually inductive types can also be propositions. For instance, we can construct the inductive predicates `occurs` and `occurs_forest` to express that a given element occurs in a tree:

```
Inductive occurs (A:Set)(a:A) : ntree A → Prop :=
  occurs_root : ∀l, occurs A a (nnode A a l)
| occurs_branches :
   ∀b l, occurs_forest A a l → occurs A a (nnode A b l)
```

```
with occurs_forest (A:Set)(a:A) : nforest A → Prop :=
  occurs_head :
   ∀t tl, occurs A a t → occurs_forest A a (ncons A t tl)
| occurs_tail :
   ∀t tl,
    occurs_forest A a tl → occurs_forest A a (ncons A t tl).
```

Here again, the induction principles that are generated by default are usually too weak and it is useful to generate the right ones with the `Scheme` command.

14.3.2 Proofs by Mutual Induction

In this section, we consider a small theorem about trees of the type `ntree`. This theorem expresses properties about two functions that compute the sum of values labeling a tree of the type `ntree` and a list of trees of the type `nforest`.

```
Fixpoint n_sum_values (t:ntree Z) : Z :=
  match t with
  | nnode z l ⇒ z + n_sum_values_l l
  end
 with n_sum_values_l (l:nforest Z) : Z :=
  match l with
  | nnil ⇒ 0
  | ncons t tl ⇒ n_sum_values t + n_sum_values_l tl
  end.
```

Our theorem expresses that the sum of all the values in a tree is greater than the number of nodes in this tree, when all the values are greater than 1.

```
Theorem greater_values_sum :
 ∀t:ntree Z,
   (∀x:Z, occurs Z x t → 1 ≤ x)→ count Z t ≤ n_sum_values t.
```

We want to prove this statement by induction over the tree `t`. Since this variable is in the type "`ntree Z`," with only one constructor that has no child of type "`ntree Z`," this proof by induction gives only an unsolvable goal if we use the default induction principle `ntree_ind`. On the other hand, the induction proof works well if we use the principle of mutual induction `ntree_ind2` that we obtained through the `Scheme` command. Using this principle requires special care, because we need to express how the variable `P0` is instantiated.

```
Proof.
 intros t; elim t using ntree_ind2 with
  (P0 := fun l:nforest Z ⇒
           (∀x:Z, occurs_forest Z x l → 1 ≤ x)→
           count_list Z l ≤ n_sum_values_l l).
```

The value given to `P0` is the "twin" proposition of the proposition in the goal statement. We simply replace occurrences of `ntree` with `nforest`, `count` with `count_list`, and so on. The induction step generates three goals. The first one is more readable after we have introduced the hypotheses and simplified the goal statement. We use the `lazy` tactic rather than `simpl` because we do not want the addition operation to be reduced as this would lead to an unreadable goal.

```
intros z l Hl Hocc; lazy beta iota delta -[Zplus Zle];
 fold count_list n_sum_values_l.
```

```
  ...
  t : ntree Z
  z : Z
  l : nforest Z
  Hl : (∀ x:Z, occurs_forest Z x l → 1 ≤ x)→
       count_list Z l ≤ n_sum_values_l l
  Hocc : ∀ x:Z, occurs Z x (nnode Z z l)→ 1 ≤ x
  ============================
   1 + count_list Z l ≤ z + n_sum_values_l l
```

Here we need a theorem that decomposes the comparison between two sums into a comparison between terms in the sum. The `SearchPattern` command finds a good one:

```
SearchPattern (_ + _ ≤ _ + _).
```

```
...
Zplus_le_compat: ∀ n m p q:Z, n ≤ m → p ≤ q → n+p ≤ m+q
```

This theorem has the right form to be used with `apply` and decomposes the goal into two subgoals that are easily proved with the hypothesis `Hocc` and the constructors of the inductive predicate `occurs`. In the second goal generated by the induction step, we have to check that the property is satisfied for empty tree lists. This goal is solved automatically. The next goal is a goal concerning empty lists of tree. It uses the predicate `P0` that we provided for the `elim` tactic.

```
 auto with *.
```

```
  ...
  t : ntree Z
  ============================
   (∀ x:Z, occurs_forest Z x (nnil Z)→ 1 ≤ x)→
   count_list Z (nnil Z) ≤ n_sum_values_l (nnil Z)
```

```
auto with zarith.
```

The next goal is the last goal generated by the induction step and corresponds to a step case on a list of trees, with the possibility of using an induction hypothesis for the first tree of the list *and* an induction hypothesis

for the tail of the list. This goal is more readable after we have introduced the variables and hypotheses in the context and simplified the goal with respect to the two functions `count_list` and `n_sum_values_l`:

```
intros t1 Hrec1 tl Hrec2 Hocc;
 lazy beta iota delta -[Zplus Zle];
 fold count count_list n_sum_values n_sum_values_l.
...
Hrec1 : (∀ x:Z, occurs Z x t1 → 1 ≤ x)→
        count Z t1 ≤ n_sum_values t1
tl : nforest Z
Hrec2 : (∀ x:Z, occurs_forest Z x tl → 1 ≤ x)→
        count_list Z tl ≤ n_sum_values_l tl
Hocc : ∀ x:Z, occurs_forest Z x (ncons Z t1 tl)→ 1 ≤ x
============================
 count Z t1 + count_list Z tl ≤ n_sum_values t1 + n_sum_values_l tl
```

Here again, the proof is easy to complete using the theorem `Zle_plus_plus` and the constructors for `occurs` and `occurs_forest`.

14.3.3 *** Trees and Tree Lists

A problem with the inductive types `ntree` and `nforest` is that the type `nforest` is only the type of lists specialized to contain trees. All functions that were already defined for lists need to be defined again for this new type and their properties need to be proved once more. This can be avoided by defining an inductive type of trees that simply relies on the type of polymorphic lists, with the following inductive definition:

```
Inductive ltree (A:Set) : Set :=
    lnode : A → list (ltree A)→ ltree A.
```

This command is accepted by *Coq*, but the induction principle constructed by default is useless. It overlooks the fact that the list included in an `lnode` term may contain subterms that deserve an induction hypothesis.

```
Check ltree_ind.
ltree_ind
    : ∀ (A:Set)(P:ltree A → Prop),
        (∀ (a:A)(l:list (ltree A)), P (lnode A a l))→
        ∀ l:ltree A, P l
```

For this kind of type, we can define a better adapted induction principle, but the `Scheme` command does not solve our problem. We have to build the induction principle by hand with the `Fixpoint` command as we did in Sect. 14.1.4. We use a nested `fix` construct. Nesting the `fix` construct inside the `Fixpoint` command ensures that the recursive calls really occur on structurally smaller terms for both recursions.

```
Section correct_ltree_ind.
Variables
  (A : Set)(P : ltree A → Prop)(Q : list (ltree A)→ Prop).

Hypotheses
 (H : ∀(a:A)(l:list (ltree A)), Q l → P (lnode A a l))
 (H0 : Q nil)
 (H1 : ∀t:ltree A, P t →
      ∀l:list (ltree A), Q l → Q (cons t l)).


Fixpoint ltree_ind2 (t:ltree A) : P t :=
  match t as x return P x with
  | lnode a l ⇒
      H a l
        (((fix l_ind (l':list (ltree A)) : Q l' :=
             match l' as x return Q x with
             | nil ⇒ H0
             | cons t1 tl ⇒ H1 t1 (ltree_ind2 t1) tl (l_ind tl)
             end)) l)
  end.
End correct_ltree_ind.
```

In this term, we call the "internal fixpoint" the expression

```
fix l_ind ... end.
```

The variable `l` is recognized as a structural subterm of `t`; the variable `l'` is also a structural subterm of `t` because it is a structural subterm of `l` through the application of the internal fixpoint. The variable `t1` is a structural subterm of `l'` and by transitivity it is also a structural subterm of `l` and `t`. Thus, applying `ltree_ind2` to `t1` is correct. This technique for building a suitable induction principle for this data structure is also applicable for writing recursive functions over trees of the type `ltree`.

While the `Scheme` command is used to provide simultaneously the induction principles for trees and forests, we need to construct manually another induction principle for tree lists, this time by nesting an anonymous structurally recursive function over trees inside a structurally recursive function over lists of trees.

Exercise 14.7 *** Build the induction principle `list_ltree_ind2` that is suitable to reason by induction over lists of trees.*

Exercise 14.8 *Define the function*

```
lcount:∀A:Set, ltree A → nat
```

that counts the number of nodes in a tree of type `ltree`.

Exercise 14.9 ***Define the functions* `ltree_to_ntree` *and* `ntree_to_ltree` *that translate trees from one type to the other and respects the structure, and prove that these functions are inverses of each other.*