11

* A Case Study

In Chap. 10, we described succinctly the principle of the extraction mechanism. This chapter contains a simple case study to illustrate the subtle links between the sorts `Prop` and `Set`. In particular, we can develop and extract certified programs that provide reasonable efficiency, thanks to our knowledge of the extraction process.

The main object of this chapter is the study of *binary search trees*. We build certified programs to find, insert, or remove data in these trees. The complete development is provided in the *Coq* user contributions.[1] Here we only present the details that are related to program extraction.

11.1 Binary Search Trees

A *binary search tree* is a binary tree where the leaves hold no information and the internal nodes are labeled—in our case with integers—with an extra condition that for every internal node labeled with some number n, the left (resp. right) subtree contains only labels that are strictly less (resp. greater) than n. An example of such a tree is given in Fig. 11.1. In our development, we do not define a *Coq* type for a binary search tree. We consider separately a data type—the type of binary trees with integer labels—and a predicate "to be a search tree" on this type. Defining this predicate requires a few auxiliary definitions.

11.1.1 The Data Structure

We already found in Sect. 6.3.4 an inductive definition of binary trees labeled with integers:

[1] Accessible at the site `http://coq.inria.fr/contribs-eng.html`; the name of this contribution is `search-trees`.

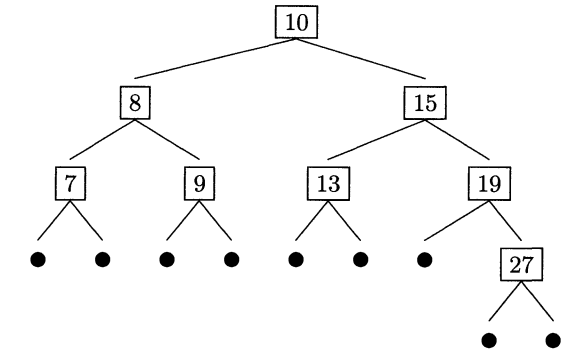

Fig. 11.1. A binary search tree

```
Open Scope Z_scope.

Inductive Z_btree : Set :=
  Z_leaf : Z_btree
| Z_bnode : Z→Z_btree→Z_btree→Z_btree.
```

For instance, the following *Gallina* expression describes the tree from Fig. 11.1:

```
Z_bnode 10
  (Z_bnode 8
    (Z_bnode 7 Z_leaf Z_leaf)
    (Z_bnode 9 Z_leaf Z_leaf))
  (Z_bnode 15
     (Z_bnode 13 Z_leaf Z_leaf)
     (Z_bnode 19 Z_leaf (Z_bnode 27 Z_leaf Z_leaf)))
```

Values Occurring in a Tree

Here is an inductive definition of the proposition "`occ` n t," meaning "the integer value n occurs at least once in the tree t":

```
Inductive occ (n:Z) : Z_btree→Prop :=
| occ_root : ∀t1 t2:Z_btree, occ n (Z_bnode n t1 t2)
| occ_l :
    ∀(p:Z)(t1 t2:Z_btree), occ n t1 → occ n (Z_bnode p t1 t2)
| occ_r :
    ∀(p:Z)(t1 t2:Z_btree), occ n t2 → occ n (Z_bnode p t1 t2).
```

11.1.2 A Naïve Approach to Deciding Occurrence

Our first objective is to develop a certified program to test whether an integer n occurs in a tree t. In other words, we want to construct a *Gallina* function with the following type:

```
∀(n:Z)(t:Z_btree), {occ n t}+{~occ n t}
```

The simplest strategy is to use a recursion on `t` and the function that decides equality on `Z`, `Z_eq_dec`. Here is the definition:[2]

```
Definition naive_occ_dec :
  ∀(n:Z)(t:Z_btree), {occ n t}+{~occ n t}.
 induction t.
 right; auto with searchtrees.
 case (Z_eq_dec n z).
 induction 1; left; auto with searchtrees.
 case IHt1; case IHt2; intros; auto with searchtrees.
 right; intro H; elim (occ_inv H); auto with searchtrees.
 tauto.
Defined.
```

With the help of the command "`Extraction naive_occ_dec`," we can observe the extracted code that corresponds to this function:

```
let rec naive_occ_dec n = function
  Z_leaf ->Right
| Z_bnode (z1, z0, z)→
    (match Z_eq_dec n z1 with
       Left ->Left
     | Right →
         (match naive_occ_dec n z0 with
            Left ->Left
          | Right ->naive_occ_dec n z))
```

This program clearly is inefficient: when n does not occur in t, the result is returned only after the tree has been completely traversed. In fact, the specification of `naive_occ_dec` prevents us from improving the algorithm. The second argument of the function is just any binary tree and we cannot avoid visiting all nodes to look for the value of the first argument in the tree. This lack of efficiency can be avoided if we restrict the occurrence test to binary trees satisfying a property that makes the complete traversal useless.

11.1.3 Describing Search Trees

We can define inductively the predicate "to be a binary search tree":

[2] The hint database `searchtrees`, specific to this development, contains a few technical lemmas that we do not detail here.

- Every leaf is a binary search tree,
- If t_1 and t_2 are binary search trees, if n is greater than every label in t_1 and less than every label in t_2, then the tree with the root label n, the left subtree t_1, and the right subtree t_2 is a binary search tree.

To formalize this in *Coq* we follow three steps:

1. defining a predicate "`min` z t" meaning "z is less than every label in t,"
2. defining a predicate "`maj` z t" meaning "z is greater than every label in t,"
3. defining inductively the predicate `search_tree:Z_btree→Prop`, using `min` and `maj` as auxiliary predicates.

Here is the *Coq* text to define these three predicates:

```
Inductive min (n:Z)(t:Z_btree) : Prop :=
    min_intro : (∀p:Z, occ p t → n < p)→ min n t.

Inductive maj (n:Z)(t:Z_btree) : Prop :=
    maj_intro : (∀p:Z, occ p t → p < n)→ maj n t.

Inductive search_tree : Z_btree→Prop :=
  | leaf_search_tree : search_tree Z_leaf
  | bnode_search_tree :
    ∀(n:Z)(t1 t2:Z_btree),
    search_tree t1 → search_tree t2 → maj n t1 → min n t2 →
    search_tree (Z_bnode n t1 t2).
```

It may look strange that `min` and `maj` are defined inductively; maybe the reader would have chosen a plain non-inductive definition:

```
Definition min (n:Z)(t:Z_btree) : Prop :=
  ∀p:Z, occ p t → n < p.
```

Nevertheless, an inductive type is more convenient, as it makes it possible to use tactics like `split` as the introduction tactic and `case` as the elimination tactic, while a plain definition forces the developer to control δ-expansion using tactics like `unfold`. A δ-expansion is difficult to control and excessive unfolding alters the readability of the goals. With an inductive type, we can construct and use propositions of the form "`min` n t" only when needed. This is a general method that we recommend.

This choice between plain definitions and inductive types is also encountered in conventional programming. For instance, we can consider the definition in *OCAML* of a type to define integer-indexed variables (for instance, in a compiler). We often prefer defining a new type:

```
type variable = Mkvar of int
```

to a simple type renaming:

```
type variable = int
```

Exercise 11.1 *** The predicates* `min` *and* `maj` *could have been directly defined as a recursive inductive predicate without using* `occ`.

Redo the whole development (from the user contribution cited above) with such a definition and compare the ease of development in the two approaches. Pay attention to minimizing the amount of modifications that are needed. This exercise shows that maintaining proofs is close to maintaining software.

11.2 Specifying Programs

The specifications of programs for finding, adding, and removing data in binary search trees are given as types of the sort `Set`. These specifications use the predicates `occ` and `search_tree`, which both live in the sort `Prop`, together with the type constructors `sig` and `sumbool` (see Sects. 9.1.1 and 9.1.3).

11.2.1 Finding an Occurrence

The problem of finding an integer p in a tree t is to produce a value indicating whether p occurs in t. We use the `sumbool` type to express the relation between the result value and the property we are looking for. Moreover, an efficient program may use the precondition that t is a search tree. We propose the following specification for the output value, a type parameterized by p and t:

```
Definition occ_dec_spec (p:Z)(t:Z_btree) : Set :=
  search_tree t → {occ p t}+{~occ p t}.
```

The program to be built has the following specification:

```
∀(p:Z)(t:Z_btree), occ_dec_spec p t
```

11.2.2 Inserting a Number

The specification of an insertion function must indicate precisely the relations between input and output. With a weak specification only stating that the result is a binary tree, we would be able to produce the tree "`Z_bnode` n t `Z_leaf`." This is not guaranteed to be a search tree.

A Predicate for Insertion

We repeat the method followed for `min` and `maj` and define an inductive predicate to express that a tree t' is obtained by inserting a value n in t, which we write "`INSERT` n t t'." This predicate condenses the following information:

- every value occurring in t must occur in t',
- the value n occurs in t',
- every value occurring in t' is either n or a value occurring in t,
- the tree t' is a search tree.

Here is the INSERT definition in *Coq*:

```
Inductive INSERT (n:Z)(t t':Z_btree) : Prop :=
  insert_intro :
  (∀p:Z, occ p t → occ p t')→ occ n t' →
  (∀p:Z, occ p t' → occ p t ∨ n = p)→ search_tree t' →
  INSERT n t t'.
```

11.2.2.1 The Specification for Insertion

The certified insertion program must map any integer n and any tree t to a tree t' and a proof of the proposition "`INSERT` n t t'." Here is the specification for the value being returned, as a type parameterized by n and t:

```
Definition insert_spec (n:Z)(t:Z_btree) : Set :=
 search_tree t → {t':Z_btree | INSERT n t t'}.
```

The specification of the program to be built is as follows:

```
∀(n:Z)(t:Z_btree), insert_spec n t
```

11.2.3 ** Removing a Number

To remove a value from a tree, the approach is similar to the approach taken for insertion. We introduce a predicate `RM` and a dependent type `rm_spec`:

```
Inductive RM (n:Z)(t t':Z_btree) : Prop :=
  rm_intro :
  ~occ n t' →
  (∀p:Z, occ p t' → occ p t)→
  (∀p:Z, occ p t → occ p t' ∨ n = p)→
  search_tree t' →
  RM n t t'.

Definition rm_spec (n:Z)(t:Z_btree) : Set :=
  search_tree t → {t' : Z_btree | RM n t t'}.
```

The specification for the remove program is as follows:

```
∀(n:Z)(t:Z_btree), rm_spec n t
```

Exercise 11.2 *** How would the specification evolve if we were to use a type of "binary search trees" in the* `Set` *sort?*

11.3 Auxiliary Lemmas

Now that we have the specifications for the three programs, it would be foolish to start developing right away. Users who follow this approach are quickly overwhelmed by the quantity of goals to solve. For instance, here is a goal that appears for the remove program:

```
...
n : Z
p : Z
t1 : Z_btree
t2 : Z_btree
t' : Z_btree
H : n < p
H0 : search_tree (Z_bnode n t1 t2)
H1 : RM p t2 t'
H2 : occ p (Z_bnode n t1 t')
D : occ p t1 ∨ occ p t'
============================
 ~occ p t1
```

Similar goals appear several times in the development and it is useful to develop a small library of technical lemmas about search trees to express the main properties. The previous goal can be solved quickly if the following lemma is established beforehand:

```
Lemma not_left :
 ∀(n:Z)(t1 t2:Z_btree),
   search_tree (Z_bnode n t1 t2)→∀p:Z, p ≥ n → ~occ p t1.
```

To get a better understanding of the needed lemmas, we give their statements in Fig. 11.2.

11.4 Realizing Specifications

When developing the certified programs, we want to avoid the drawback of our first trial presented in Sect. 11.1.2; we should not let automatic proof procedures decide on the exact algorithms. We prefer to guide the construction of the certified program using the `refine` tactic introduced in Sect. 9.2.7.

11.4.1 Realizing the Occurrence Test

Recall that we want to build a term for the following specification:

```
Definition occ_dec : ∀(p:Z)(t:Z_btree), occ_dec_spec p t.
```

```
Lemma min_leaf : ∀z:Z, min z Z_leaf.

Lemma maj_leaf : ∀z:Z, maj z Z_leaf.

Lemma maj_not_occ : ∀(z:Z)(t:Z_btree), maj z t → ~occ z t.

Lemma min_not_occ : ∀(z:Z)(t:Z_btree), min z t → ~occ z t.

Section search_tree_basic_properties.
  Variable n : Z.
  Variables t1 t2 : Z_btree.
  Hypothesis se : search_tree (Z_bnode n t1 t2).

  Lemma search_tree_l : search_tree t1.

  Lemma search_tree_r : search_tree t2.

  Lemma maj_l : maj n t1.

  Lemma min_r : min n t2.

  Lemma not_right : ∀p:Z, p ≤ n → ~occ p t2.

  Lemma not_left : ∀p:Z, p ≥ n → ~occ p t1.

  Lemma go_left :
   ∀p:Z, occ p (Z_bnode n t1 t2)→ p < n → occ p t1.

  Lemma go_right :
   ∀p:Z, occ p (Z_bnode n t1 t2)→ p > n → occ p t2.

End search_tree_basic_properties.

Hint Resolve go_left  go_right  not_left not_right
  search_tree_l search_tree_r maj_l min_r : searchtrees.
```

Fig. 11.2. Technical lemmas on search trees

We can easily get rid of the simplest case, where the tree is only a leaf; the integer `p` cannot occur and the `right` constructor of the `sumbool` type is appropriate.

In the general case, where the tree has the form "`Z_bnode` n t_1 t_2," we need to compare n and p to decide if we want to give a positive answer because $n = p$, or look inside t_1, or look inside t_2. Two functions are provided in the `ZArith` library to express that the order on `Z` is total and decidable:

```
Z_le_gt_dec : ∀x y:Z, {x ≤ y}+{x > y}
```

`Z_le_lt_eq_dec : ∀x y:Z, x ≤ y → {x < y}+{x = y}`

With these functions, we distinguish three cases:

1. If $p < n$, we need a recursive call of the form "`occ_dec` p t_1":
   - if "`occ_dec` p t_1" returns "`left _ _` π," then the term π is a proof of "`occ` p t_1" and the program should return "`left _ _` π'" where π' is a proof of "`occ` p `(Z_bnode` n t_1 t_2`)`,"
   - if "`occ_dec` p t_1" returns "`right _ _` π," then the term π is a proof of "$\sim$`occ` p t_1" and the program should return "`right _ _` π'" where π' is a proof of "$\sim$`occ` p `(Z_bnode` n t_1 t_2`)`."
2. If $p = n$, the program should return "`left _ _` π" where π is a proof of "`occ` p `(Z_bnode` n t_1 t_2`)`."
3. If $p > n$, a symmetric approach to the case $p < n$ should be taken.

With the `refine` tactic, we give a term where the logical parts are left unknown and represented by jokers, most of these jokers are solved either by `refine` or by automatic tactics using the technical lemmas in the `searchtrees` database:

```
Definition occ_dec : ∀(p:Z)(t:Z_btree), occ_dec_spec p t.
 refine
  (fix occ_dec (p:Z)(t:Z_btree){struct t} : occ_dec_spec p t :=
    match t as x return occ_dec_spec p x with
    | Z_leaf ⇒ fun h ⇒ right _ _
    | Z_bnode n t1 t2 ⇒
      fun h ⇒
        match Z_le_gt_dec p n with
        | left h1 ⇒
          match Z_le_lt_eq_dec p n h1 with
          | left h'1 ⇒
            match occ_dec p t1 _ with
            | left h''1 ⇒ left _ _
            | right h''2 ⇒ right _ _
            end
          | right h'2 ⇒ left _ _
          end
        | right h2 ⇒
          match occ_dec p t2 _ with
          | left h''1 ⇒ left _ _
          | right h''2 ⇒ right _ _
          end
        end
    end); eauto with searchtrees.
 rewrite h'2; auto with searchtrees.
Defined.
```

With the command "`Extraction occ_dec`" we can observe the algorithmic content of our development in *OCAML* syntax:

```
let rec occ_dec p = function
| Z_leaf -> Right
| Z_bnode (n, t1, t2) ->
  (match z_le_gt_dec p n with
   | Left ->
     (match z_le_lt_eq_dec p n with
        Left -> occ_dec p t1
      | Right -> Left)
   | Right -> occ_dec p t2)
```

Recall that the constructors `left` and `right` should be assimilated to `true` and `false`; with more conventional syntax, this program could be rephrased as follows:

```
let rec occ_dec p t =
 match t with
  Z_leaf -> false
| Z_bnode(n,t1,t2) ->
    if p <= n
    then  if p < n then occ_dec p t1 else true
    else occ_dec p t2
```

The program we obtain goes down only one branch of the binary search tree, guided by the comparisons with the successive labels down the branch.

11.4.2 Insertion

The approach to insert an integer in a binary search tree is very close to the approach for the occurrence test. We show only the most important differences.

We prove a collection of lemmas about the `INSERT` predicate that are similar to the clauses that a *Prolog* programmer would write to define this predicate. The statements of these lemmas are given in Fig. 11.3 (without the proofs). Adding these four lemmas to the `searchtrees` database for `auto` also significantly helps the description of the certified program using tactics as given in Fig. 11.4.

In the *Coq* version, the three arguments of the `insert` function have types that belong to the `Set` sort and the `Prop` sort. The function's output is a pair where one part is in the `Set` sort and the other part is in the `Prop` sort. The distinction between these sorts plays an important role in controlling the amount of computation that takes place in the extracted function. In the *Coq* version of the function, there is some computation included in the function to build the proof part of the output. At extraction time, the proof argument to the function is dropped and so is the proof part in the function's

```
Lemma insert_leaf :
 ∀n:Z, INSERT n Z_leaf (Z_bnode n Z_leaf Z_leaf).

Lemma insert_l :
 ∀(n p:Z)(t1 t'1 t2:Z_btree),
   n < p →
   search_tree (Z_bnode p t1 t2)→
   INSERT n t1 t'1 →
   INSERT n (Z_bnode p t1 t2)(Z_bnode p t'1 t2).

Lemma insert_r :
 ∀(n p:Z)(t1 t2 t'2:Z_btree),
   n > p →
   search_tree (Z_bnode p t1 t2)→
   INSERT n t2 t'2 →
   INSERT n (Z_bnode p t1 t2)(Z_bnode p t1 t'2).

Lemma insert_eq :
 ∀(n:Z)(t1 t2:Z_btree), search_tree (Z_bnode n t1 t2)→
   INSERT n (Z_bnode n t1 t2)(Z_bnode n t1 t2).

Hint Resolve insert_leaf insert_l insert_r insert_eq
  : searchtrees.
```

Fig. 11.3. *Prolog*-like lemmas for insertion

output, along with the computation that was needed to build this proof. Even though strongly specified functions seem to contain more computation than weakly specified functions, their extracted counterpart can be as efficient if the designer was careful to ensure that only the relevant computation is placed in the `Set` sort.

Should There be a Test Function for `search_tree`?

Most of our lemmas and certified program use a hypothesis or precondition stating the property "`search_tree` t." The predicate `search_tree` has type `Z_btree→Prop` and cannot be confused with a function returning a boolean value that could be used inside programs. We could develop a decision procedure for this predicate that would have the following type:

```
search_tree_dec : ∀t:Z_btree, {search_tree t}+{~search_tree t}
```

This function could then be used to filter the trees, on which our functions could be applied. In our approach, we do not need such a function; it is more natural that we only use trees that are obtained from the empty tree by successively inserting new elements. For instance, we can specify a function that constructs a binary search tree from a list of numbers, so that the tree contains exactly the elements of the list:

```
Definition insert : ∀(n:Z)(t:Z_btree), insert_spec n t.
 refine
  (fix insert (n:Z)(t:Z_btree){struct t}
     : insert_spec n t :=
     match t return insert_spec n t with
     | Z_leaf ⇒
       fun s ⇒ exist _ (Z_bnode n Z_leaf Z_leaf) _
     | Z_bnode p t1 t2 ⇒
       fun s ⇒
        match Z_le_gt_dec n p with
        | left h ⇒
          match Z_le_lt_eq_dec n p h with
          | left _ ⇒
            match insert n t1 _ with
            | exist t3 _ ⇒ exist _ (Z_bnode p t3 t2) _
            end
          | right h' ⇒ exist _ (Z_bnode n t1 t2) _
          end
        | right _ ⇒
          match insert n t2 _ with
          | exist t3 _ ⇒ exist _ (Z_bnode p t1 t3) _
          end
        end
     end); eauto with searchtrees.
 rewrite h'; eauto with searchtrees.
Defined.
```

Fig. 11.4. Describing the insertion program

```
Definition list2tree_spec (l:list Z) : Set :=
  {t : Z_btree | search_tree t ∧ (∀p:Z, In p l ↔ occ p t)}.
```

With this function, we can build large binary search trees, without ever checking whether insertions happen on a binary search tree, because this is given by the typing constraints.

To develop this program converting lists of values into binary search trees, we follow a classical approach of functional programming, where the main function relies on an auxiliary terminal recursive function.

The specification for this auxiliary function is that it takes a list l and a tree t and returns a tree t' that contains exactly the union of elements from l and t:

```
Definition list2tree_aux_spec (l:list Z)(t:Z_btree) :=
  search_tree t →
 {t' : Z_btree | search_tree t' ∧
                 (∀p:Z, In p l ∨ occ p t ↔ occ p t')}.
```

We again use `refine` to propose a realization for these specifications. The term given as the argument to `refine` is quite complex, so we advise the users

to decompose this term into fragments during interactive experiments with the *Coq* system.

```
Definition list2tree_aux :
  ∀(l:list Z)(t:Z_btree), list2tree_aux_spec l t.
refine
 (fix list2tree_aux (l:list Z) :
   ∀t:Z_btree, list2tree_aux_spec l t :=
   fun t ⇒
     match l return list2tree_aux_spec l t with
     | nil ⇒ fun s ⇒ exist _ t _
     | cons p l' ⇒
       fun s ⇒
         match insert p (t:=t) s with
         | exist t' _ ⇒
           match list2tree_aux l' t' _ with
           | exist t'' _ ⇒ exist _ t'' _
           end
         end
     end).
...
Defined.


Definition list2tree : ∀l:list Z, list2tree_spec l.
 refine
  (fun l ⇒ match list2tree_aux l (t:=Z_leaf) _ with
                    | exist t _ ⇒ exist _ t _
                    end).
...
Defined.
```

Extracted Programs

The extracted programs for the functions `insert` and `list2tree` are very simple. Parent [69], Filliâtre [40] and Balaa and Bertot [6] studied ways to guide the construction of the *Coq* certified program with fewer details. There is hope that future work will make the description of algorithms simpler.

```
let rec insert n = function
| Z_leaf -> Z_bnode (n, Z_leaf, Z_leaf)
| Z_bnode (p, t1, t2) ->
  (match z_le_gt_dec n p with
   | Left ->
     (match z_le_lt_eq_dec n p with
      | Left -> Z_bnode (p, (insert n t1), t2)
```

```
      | Right -> Z_bnode (n, t1, t2))
   | Right -> Z_bnode (p, t1, (insert n t2)))

let rec list2tree_aux l t =
  match l with
    | Nil -> t
    | Cons (p, l') -> list2tree_aux l' (insert p t)

let list2tree l =
  list2tree_aux l Z_leaf
```

11.4.3 Removing Elements

Removing an element in a binary search tree is not much more complex than inserting one, except when the value to remove labels the root of the tree. The usual solution to remove n from "`Z_bnode` n t_1 t_2" is to remove the greatest element q from t_1, thus obtaining a result tree r, and to build and return the new tree "`Z_bnode` q r t_2"; when t_1 is a leaf, one simply returns t_2.

From the programming point of view, we see that there is again a need for an auxiliary function, here with the specification that it removes the largest element from a non-empty tree. We proceed as in the previous sections, by defining an inductive predicate `RMAX` that describes the relation between the input and output of the auxiliary function:

```
Inductive RMAX (t t':Z_btree)(n:Z) : Prop :=
  rmax_intro :
  occ n t →
  (∀p:Z, occ p t → p ≤ n)→
  (∀q:Z, occ q t' → occ q t)→
  (∀q:Z, occ q t → occ q t' ∨ n = q)→
  ~occ n t' → search_tree t' → RMAX t t' n.
```

Again, we prove a collection of *Prolog*-like lemmas for this predicate and we give the specification for the auxiliary function. Note that we use the inductive type `sigS` introduced in Sect. 9.1.2, because the function returns two pieces of data:

```
Definition rmax_sig (t:Z_btree)(q:Z) :=
  {t':Z_btree | RMAX t t' q}.

Definition rmax_spec (t:Z_btree) :=
  search_tree t → is_bnode t → {q:Z &  rmax_sig t q}.

Definition rmax : ∀t:Z_btree, rmax_spec t.
```

We do not give the details of how the function `rmax` is described, but we prsent the code that is obtained after extracting `rmax` and the main function `rm`:

```
let rec rmax = function
| Z_leaf -> assert false (* absurd case *)
| Z_bnode (r, t1, t2) ->
  (match t2 with
   | Z_leaf -> ExistS (r, t1)
   | Z_bnode (n', t'1, t'2) ->
     let ExistS (num, r0) = rmax t2 in
       ExistS (num, (Z_bnode (r, t1, r0))))

let rec rm n = function
| Z_leaf -> Z_leaf
| Z_bnode (p, t1, t2) ->
  (match z_le_gt_dec n p with
   | Left ->
     (match z_le_lt_eq_dec n p with
      | Left -> Z_bnode (p, (rm n t1), t2)
      | Right ->
        (match t1 with
         | Z_leaf -> t2
         | Z_bnode (p', t'1, t'2) ->
           let ExistS (q, r) = rmax (Z_bnode (p', t'1, t'2)) in
             Z_bnode (q, r, t2)))
   | Right -> Z_bnode (p, t1, (rm n t2)))
```

Note that the function `rmax` contains an expression "`assert false`" that corresponds to a contradictory case. We have a precondition "`is_bnode t`" but the pattern matching construct has a rule for the case where `t` is a leaf that is useless.

11.5 Possible Improvements

We can only represent finite sets of integers with the binary search trees presented so far. The only property of `Z` that we used is the fact that the relation $\leq$ is a decidable total order. It should be possible to generalize our approach to every type and order that have these characteristics,namely `nat`, `Z*Z`, "`list bool`," and so on.

Another use of binary search trees is to have them represent partial functions with a finite domain. We cannot use our toy implementation for this purpose. The next chapter studies the module system in *Coq*, a functionality that makes it possible to reuse a development like this one for several different types. The example used to illustrate this module system is the representation of partial functions with a finite domain in a type with a decidable total order.