Functors for Proofs and Programs

Jean-Christophe Filliâtre and Pierre Letouzey

LRI – CNRS UMR 8623
Université Paris-Sud, France
{filliatr,letouzey}@lri.fr

September 2003

Abstract

This paper presents the formal verification with the Coq proof assistant of several applicative data structures implementing finite sets. These implementations are parameterized by an ordered type for the elements, using functors from the ML module system. The verification follows closely this scheme, using the newly Coq module system. One of the verified implementation is the actual code for sets and maps from the Objective Caml standard library. The process of verification exhibited two small errors in the balancing scheme, which have been fixed and then verified. Beyond these verification results, this article illustrates the use and benefits of modules and functors in a logical framework.

1 Introduction

Balanced trees are notoriously hard to implement without any mistake. Exact invariants are difficult to figure out, even for applicative implementations. Since most programming languages provide data structures for finite sets and dictionaries based on balanced trees, this is a real challenge for formal verification to actually verify one of these.

We choose to verify the `Set` module from the Objective Caml (OCaml) standard library [2]. This applicative implementation of finite sets uses AVL trees [4] and provides all expected operations, including union, difference and intersection. Above all, this is a very efficient and heavily used implementation, which motivates a correctness proof. This article also presents the verification of two other implementations, using respectively sorted lists and red-black trees [8], both written in OCaml and using the same interface as `Set`.

Building balanced trees over values of a given type requires this type to be equipped with a total ordering function. Several techniques are available to build a parametric library: in ML, polymorphism gives genericity over the type and first-class functions give the genericity over the ordering function (e.g. it is passed when initially creating the tree); in object oriented languages, objects to be stored in the trees are given a suitable comparison function; etc.

The most elegant technique is probably the one provided by the ML module system, as implemented in SML [9] and OCAML [11]. A *module* is a collection of definitions of types, values and submodules. Its type is called a *signature* and contains declarations of (some of the) types, values and submodules. Modules can be parameterized by some signatures and later applied to actual modules. Such functions from modules to modules are called *functors*. The `Set` library is a nice illustration of OCAML module system. It is naturally written as a functor taking the signature of an ordered type as argument and returning the signature for finite sets as a result.

The use of modules and functors introduces an extra challenge for the formal proof. The recent introduction of an ML-like module system into the COQ proof assistant [1, 5] makes it a perfect candidate for this verification. As a side effect, this article examplifies the use of modules and functors in a logical framework, which goes well beyond its use in programming languages.

Currently, COQ is not able to reason directly about OCAML code. To certify an OCAML applicative implementation, we first translate it into COQ own programming language, GALLINA. Then the logic of COQ can be used to express properties of the GALLINA functions. Since the translation from OCAML to GALLINA is done manually and is thus error-prone, COQ provides an automated mechanism for the converse translation called *extraction*. The extraction takes a COQ function or proof, removes all its logical statements and translates the remaining algorithmic content to OCAML. We finally end up with three versions of the code: the OCAML original handwritten one, its GALLINA translation certified in COQ, and the OCAML extracted version. Theoretical results about the extraction [12, 13] ensure that the extracted code verifies the same properties as the GALLINA version. In practice, the extracted code behaves reasonably well (see e.g. the final benchmarks) and often presents only syntactical differences with the original OCAML code.

Outline. This paper is organized as follows. Section 2 is devoted to the presentation of OCAML and COQ module systems. Section 3 introduces the signatures for ordered types and finite sets, and various utility functors over these signatures. Section 4 presents the verification of three finite sets implementations, using respectively sorted lists, AVL trees from the OCAML standard library and red-black trees. Section 5 concludes with a benchmark comparing performances of handwritten and extracted code.

Source code. This article only details the most important parts of the formal development. The whole source code would take too much space, even in appendix: the sole specification is 2338 lines long and the proof scripts amount to 5704 lines, not mentioning the original source code for red-black and AVL trees. All these files are available at `http://www.lri.fr/~filliatr/fsets/` for downloading and browsing. The COQ files need the development version of COQ to compile and a rather powerful machine: 10 minutes are indeed necessary to compile the whole development on a 1 Gigahertz Intel CPU.

The pieces of COQ and OCAML code given in this article are displayed in `verbatim` font, apart from the embellishment of a few symbols: $\rightarrow$ for `->`, $\leftrightarrow$ for `<->`, $\vee$ for `\/`, $\wedge$ for `/\` and $\neg$ for `~`.

2 Modules and functors

This section introduces the OCAML and COQ module systems.

2.1 The OCAML module system

The OCAML module system [11] is derived from the original one for SML [9]. The latter also evolved in return, both systems being now quite close and known as the Harper-Lillibridge-Leroy module system. This section briefly illustrates the OCAML module system with the `Set` library from its standard library, which signature is used throughout this paper and which code is verified in Section 4.2.

The `Set` library first defines a signature `S` for finite sets, given Figure 1. It contains the type `elt` of elements, the type `t` of sets, the value `empty` for the empty set, and 22 operations over sets. Most of them have an obvious meaning and the expected semantics. Other like `fold`, `elements` or `choose` have part of their specification left unspecified (this is detailed later in this paper).

The set implementation is parameterized by the type of its elements, which must be equipped with a total ordered function. The following signature `OrderedType` is introduced for this purpose:

```
module type OrderedType =  sig
  type t
  val compare : t → t → int
end
```

The `compare` function is returning an integer, such that `(compare x y)` is zero if `x` and `y` are equal, negative if `x` is smaller than `y` and positive if `x` is greater than `y`. For instance, a module `Int` realizing this signature for the type `int` of integers and the predefined comparison function `Pervasives.compare`[1] would be:

```
module Int : Set.OrderedType = struct
  type t = int
  let compare = Pervasives.compare
end
```

The implementation of the data structure for sets is provided as a functor `Make` taking a module `Ord` of signature `OrderedType` as argument and returning a module of signature `S`:

```
module Make (Ord : OrderedType) : S with type elt = Ord.t
```

The signature for the returned module needs to be more precise than `S`: we must identify the type `elt` in the returned signature with the type `Ord.t` given as argument. This is made explicit using the `with type` construct.

We would get sets of integers by applying this functor to the module `Int` above:

[1]The subtraction can not be used as comparison function for integers because of overflows; consider for instance `min_int` $-$ `max_int` $= 1$.

```
module type S = sig
  type elt
  type t
  val empty : t
  val is_empty : t → bool
  val mem : elt → t → bool
  val add : elt → t → t
  val singleton : elt → t
  val remove : elt → t → t
  val union : t → t → t
  val inter : t → t → t
  val diff : t → t → t
  val compare : t → t → int
  val equal : t → t → bool
  val subset : t → t → bool
  val iter : (elt → unit) → t → unit
  val fold : (elt → 'a → 'a) → t → 'a → 'a
  val for_all : (elt → bool) → t → bool
  val exists : (elt → bool) → t → bool
  val filter : (elt → bool) → t → t
  val partition : (elt → bool) → t → t * t
  val cardinal : t → int
  val elements : t → elt list
  val min_elt : t → elt
  val max_elt : t → elt
  val choose : t → elt
end
```

Figure 1: OCAML signature `Set.S` for finite sets

```
module IntSet = Set.Make(Int)
```

It is important to notice that signature `S` contains a type `t` and a comparison function `compare` (with the expected behavior), allowing to build sets of sets by applying `Make` again on the resulting module. For instance, sets of sets of integers would be obtained with:

```
module IntSetSet = Set.Make(IntSet)
```

The reader may refer to the OCAML manual [2] for further details on its module system.

2.2 The COQ module system

A module system for the COQ proof assistant has been advocated for a long time, mainly in Courant's PhD thesis [7]. Clearly more ambitious than OCAML's, his module system appeared too complex to be implemented in the existing code of COQ. Instead, a module system *à la Caml* was recently implemented by Chrzaszcz [5] following Leroy's modular module system [11].

The main difference with OCAML is that COQ modules are *interactive*. While in OCAML a module is introduced with a single declaration, COQ allows to build it piece by piece. The following statement starts the declaration of a module M:

```
Module M.
```

Then any subsequent COQ declaration (type, function, lemma, theorem, ...) is placed in that module, until the closing of M with

```
End M.
```

This interactive process is plainly justified by the interactive building of (most) COQ proofs. Checking a whole module with all its proofs one at a time would be intractable. Similarly, a signature S is interactively declared with

```
Module Type S.
...
End S.
```

and a functor F with

```
Module F [X : S].
...
End F.
```

The contents of these modules, signatures and functors are regular COQ declarations (inductive types, definitions, ...). In particular, modules and functors may contain logical properties and their proofs. This is of course a novelty when compared with OCAML.

A signature may contain declarations like `Parameter x : T`, claiming the existence of a value `x` of type `T` in any module implementing this signature. Depending on `T`, it corresponds either to an OCAML abstract type or to an OCAML value.

2.3 Extraction

The COQ extraction mechanism [12, 13] handles naturally the module system: COQ modules are extracted to OCAML modules, COQ signatures to OCAML signatures and COQ functors to OCAML functors.

The first task of the extraction is to introduce a distinction between terms and types, since COQ has a unified notion of terms. For instance, a `Parameter` declaration may either be extracted to a type declaration or to a value declaration, depending on its type.

Then the extraction removes all the logical parts that "decorate" the computationally relevant terms: logical justification, pre-conditions, post-conditions, ... These logical parts are not needed for the actual computation. The detection of these logical parts is done accordingly to the COQ sorts. A fundamental principle of COQ is that any COQ term belongs either to the logical sort `Prop`, or to the informative sorts `Set` and `Type`. The extraction follows this duality to decide which (sub-)terms must be erased.

3 Specifying a finite sets library

This section introduces the various signatures and functors related to the Coq specification of the finite sets library.

3.1 Ordered types

Similarly to OCaml code, our implementations are parameterized by the type of elements and its total ordering function. The OCaml ordering functions return integers for efficiency reasons. The Coq ordering functions could simply return into a three values enumeration type, such as

```
Inductive Compare : Set := Lt : Compare | Eq : Compare | Gt : Compare.
```

However, it is more idiomatic to introduce two predicates `lt` and `eq`—`lt` for the strict order relation and `eq` for the equality—and to have constructors for the inductive type `Compare` carrying proofs of the corresponding relations. Since this type is going to be reused at several places, we make it polymorphic with respect to the type `X` of elements and to the relations `lt` and `eq`:

```
Inductive Compare [X:Set; lt,eq:X→X→Prop; x,y:X] : Set :=
  | Lt : (lt x y) → (Compare lt eq x y)
  | Eq : (eq x y) → (Compare lt eq x y)
  | Gt : (lt y x) → (Compare lt eq x y).
```

Note that this type is in sort `Set` while `lt` and `eq` are in sort `Prop`. Thus the informative contents of `Compare` is a three constant values type, with the same meaning as integers in OCaml comparison functions.

Then we introduce a signature `OrderedType` for ordered types. First, it contains a type `t` for the elements, which is clearly informative and thus in sort `Set`. Then it equips the type `t` with a an equality `eq` and a strict order relation `lt`, together with a decidability function `compare` such that `(compare x y)` has type `(Compare lt eq x y)`[2].

Finally, it contains a minimal set of logical properties (`eq_refl`,...,`lt_not_eq`) expressing that `eq` is an equivalence relation and that `lt` is a strict order relation compatible with `eq`. Note that `compare` is providing the totality of this order relation. The final signature `OrderedType` is given Figure 2.

Many additional properties can be derived from this set of axioms, such as the antisymmetry of `lt`:

```
  (x,y:t)(lt x y) → ¬(lt y x)
```

Such properties are clearly useful for the forthcoming proof developments. Instead of polluting the `OrderedType` signature with all of them, we build a functor that derives these properties from `OrderedType`:

[2]Most type arguments, such as `X` in this term, are automatically inferred by Coq and thus omitted.

```
Module Type OrderedType.
  Parameter t : Set.
  Parameter eq : t → t → Prop.
  Parameter lt : t → t → Prop.
  Parameter eq_refl : (x:t) (eq x x).
  Parameter eq_sym : (x,y:t) (eq x y) → (eq y x).
  Parameter eq_trans : (x,y,z:t) (eq x y) → (eq y z) → (eq x z).
  Parameter lt_trans : (x,y,z:t) (lt x y) → (lt y z) → (lt x z).
  Parameter lt_not_eq : (x,y:t) (lt x y) → ¬(eq x y).
  Parameter compare : (x,y:t)(Compare lt eq x y).
End OrderedType.
```

Figure 2: Coq signature for ordered types

```
Module OrderedTypeFacts [O:OrderedType].
  Lemma lt_not_gt : (x,y:O.t)(O.lt x y) → ¬(O.lt y x).
  Proof.
   Intros; Intro; Absurd (O.eq x x); EAuto.
  Qed.
  (* 22 other lemmas, 3 notations, 3 tactic definitions *)
  (* and a few hints for the Auto tactic *)
End OrderedTypeFacts.
```

This way, the user only has to implement a minimal signature and all the remaining is automatically derived.

3.2 Finite sets

Reproducing the Ocaml `Set.S` signature in Coq is not immediate. First of all, four functions are not applicative. One is `iter`, which iterates a side-effects-only function over all elements of a set; we simply discard this function. The others are `min_elt`, `max_elt` and `choose`, which respectively return the minimum, the maximum and an arbitrary element of a given set, and raise an exception when the set is empty; we slightly change their type so that they now return into a sum type.

Then we face the problem of specifications: they are given as comments in the Ocaml files, and are more or less precise. Sometimes the behavior is even left partly unspecified, as for `fold`. The remaining of this section explains how these informal specifications are translated into Coq.

Finally, we face a question of style. There are indeed several ways of defining, specifying and proving correct a function in Coq. Basically, this can be done in two steps—we first define a purely informative function and then we prove it correct—or in a single one—we use dependent types to have the function returning its result together with the proof that it is correct. Our formalization actually provides both styles, with bridge functors to go from one to the other, allowing the user to choose what he or she considers as the most convenient.

3.2.1 Non-dependent signature

We first introduce a signature S containing purely informative functions together with axioms. It is very close to the OCAML signature. It introduces an ordered type for the elements:

```
Module Type S.
  Declare Module E : OrderedType.
  Definition elt := E.t.
```

Then it introduces an abstract type t for sets:

```
  Parameter t : Set.
```

All operations have exactly the same types as in OCAML (see Figure 1):

```
  Parameter empty : t.
  Parameter mem : elt → t → bool.
  Parameter add : elt → t → t.
  ...
```

apart from min_elt, max_elt and choose:

```
  Parameter min_elt : t → (option elt).
  Parameter max_elt : t → (option elt).
  Parameter choose : t → (option elt).
```

and compare which uses the Compare type introduced in Section 3.1 and refers to two predicates eq and lt also declared in the interface:

```
  Parameter eq : t → t → Prop.
  Parameter lt : t → t → Prop.
  Parameter compare : (s,s':t)(Compare lt eq s s').
```

The five properties of eq and lt are declared, making this interface a subtype of OrderedType and thus allowing a bootstrap to get sets of sets[3]:

```
  Parameter eq_refl : (eq s s).
  Parameter eq_sym : (eq s s') → (eq s' s).
  Parameter eq_trans : (eq s s') → (eq s' s'') → (eq s s'').
  Parameter lt_trans : (lt s s') → (lt s' s'') → (lt s s'').
  Parameter lt_not_eq : (lt s s') → ¬(eq s s').
```

To specify all the operations, a membership predicate is introduced:

```
  Parameter In : elt → t → Prop.
```

[3]In the remaining of this signature, free variables such as x, y, s, etc. are universally quantified.

We could have used the `mem` operation for this purpose, using `(mem x s)=true` instead of `(In x s)`, but it is more idiomatic in Coq to use propositions rather than boolean functions[4]. Moreover, it gives the implementor the opportunity to define a membership predicate without referring to `mem`, which may ease the correctness proof. An obvious property of `In` with respect to the equality `E.eq` is declared:

```
Parameter eq_In : (E.eq x y) → (In x s) → (In y s).
```

Operations are then axiomatized using the membership predicate. The value `empty` and the operations `mem`, `is_empty`, `add`, `remove`, `singleton`, `union`, `inter`, `diff`, `equal`, `subset`, `elements`, `min_elt`, `max_elt` and `choose` have obvious specifications. Here are for instance the specification of the empty set `empty`:

```
Parameter empty_1 : (a:elt)¬(In a empty).
```

and of the insertion `add`:

```
Parameter add_1 : (E.eq y x) → (In y (add x s)).
Parameter add_2 : (In y s) → (In y (add x s)).
Parameter add_3 : ¬(E.eq x y) → (In y (add x s)) → (In y s).
```

The remaining six operations, namely `filter`, `cardinal`, `fold`, `for_all`, `exists` and `partition`, are not so simple to specify and deserve a bit of explanation.

Filtering and test functions. In the OCaml standard library, the `filter` operation is specified as follows:

```
val filter : (elt → bool) → t → t
(** filter p s returns the set of all elements in s that satisfy
    predicate p. *)
```

This is slightly incorrect, as the predicate `p` could return different values for elements identified by the equality `E.eq` over type `elt`. The predicate `p` has to be *compatible* with `E.eq` in the following way:

```
Definition compat_bool [p:elt→bool] := (x,y:elt)(E.eq x y) → (p x)=(p y).
```

Then `filter` can be formally specified with the following three axioms:

```
Parameter filter_1 : (compat_bool p) → (In x (filter p s)) → (In x s).
Parameter filter_2 : (compat_bool p) → (In x (filter p s)) → (p x)=true.
Parameter filter_3 :
  (compat_bool p) → (In x s) → (p x)=true → (In x (filter p s)).
```

Note that it leaves the behavior of `filter` unspecified whenever `p` is not compatible with `E.eq`. Operations `for_all`, `exists` and `partition` are specified in a similar way.

[4]Contrary to other systems like PVS, Coq does *not* identify propositions and booleans.

Folding and cardinal. Specifying the `fold` operation poses another challenge. The OCAML specification reads:

```
val fold : (elt → 'a → 'a) → t → 'a → 'a
(** fold f s a computes (f xN ... (f x2 (f x1 a))...), where x1... xN
    are the elements of s. The order in which elements of s are
    presented to f is unspecified. *)
```

To resemble the OCAML specification as much as possible, we declare the existence of a list of elements without duplicate—the list `x1`,..., `xN` above—and we reuse the existing folding operation `fold_right` over lists:

```
Parameter fold_1 :
 (A:Set)(i:A)(f:elt→A→A)
 (EX l:(list elt) |
     (Unique E.eq l) ∧
     ((x:elt)(In x s) ↔ (InList E.eq x l)) ∧
     (fold f s i) = (fold_right f i l)).
```

`Unique` is a predicate expressing the uniqueness of elements within a list with respect to a given equality, here `E.eq`. The `cardinal` operation is specified in a similar way, using the operation `length` over lists:

```
Parameter cardinal_1 :
  (EX l:(list elt) |
      (Unique E.eq l) ∧
      ((x:elt)(In x s) ↔ (InList E.eq x l)) ∧
      (cardinal s) = (length l)).
```

Note that `cardinal` could be defined with `fold`; this will be discussed later in Section 3.3.

3.2.2 Dependent signature

We introduce a second signature for finite sets, `Sdep`. It makes use of dependent types to mix computational and logical contents, in a COQ idiomatic way of doing. The part of the signature related to the type `t` and the relations `eq`, `lt` and `In` is exactly the same as for signature `S`. Then each operation is introduced and specified with a single declaration. For instance, the empty set is declared as follows:

```
Parameter empty : { s:t | (a:elt)¬(In a s) }.
```

which must read "there exists a set `s` such that ...". Similarly, the operation `add` is declared as:

```
Parameter add : (x:elt) (s:t)
                { s':t | (y:elt)(In y s') ↔ ((E.eq y x) ∨ (In y s)) }.
```

and so on for all other operations. Only the four operations involving a predicate over elements—namely filter, for_all, exists and partition—have a slightly different specification. Indeed, to be consistent with the use of dependent types, they do not take a boolean predicate as argument but require instead a predicate in sort Prop together with a proof that it is decidable. Here is for instance the specification of filter:

```
  Parameter filter : (P:elt→Prop)(Pdec:(x:elt){P x}+{¬(P x)})(s:t)
     { s':t | (compat_P E.eq P) → (x:elt)(In x s') ↔ ((In x s)∧(P x)) }.
```

which must read "for any predicate P, any decision procedure Pdec for P and any set s, there exists a set s' such that ...". compat_P expresses the compatibility of P with respect to equality E.eq, in a way similar to compat_bool in signature S.

3.2.3 Bridge functors

Signatures S and Sdep can be proved equivalent in a constructive way. Indeed, we can implement two *bridge* functors between the two signatures. The first one is implementing the signature Sdep given a module implementing the signature S:

```
Module DepOfNodep [M:S] <: Sdep with Module E := M.E.
...
End DepOfNodep.
```

and the second one is implementing the signature S given a module implementing the signature Sdep:

```
Module NodepOfDep [M:Sdep] <: S with Module E := M.E.
...
End NodepOfDep.
```

The practical interest is obvious: the user may prefer one style of programming/proving with Coq while a particular implementation of finite sets is provided with the other style. Applying the appropriate functor is providing the desired interface.

3.3 Additional properties

Signatures S and Sdep intend to be minimal. Many additional properties can be derived. They may involve the set operations separetely or together, as in the following fact:

$$\texttt{cardinal}\ (\texttt{union a b}) + \texttt{cardinal}\ (\texttt{inter a b}) = \texttt{cardinal a} + \texttt{cardinal b}$$

Similarly to what we did for OrderedType in Section 3.1, we gather all such properties in a functor taking a module of signature S as argument:

```
Module Properties [M:S].
  Lemma union_inter_cardinal :
    (a,b:t)(cardinal (union a b)) + (cardinal (inter a b))
         = (cardinal a) + (cardinal b).
  Proof.
  ...
End Properties.
```

4 Verifying finite sets implementations

Implementing and verifying a set library with all operations introduced so far is not necessarily difficult: indeed, all operations can be coded using the four *primitive* operations `empty`, `add`, `remove` and `fold`. However, most operations can be coded more efficiently in a direct way, at the extra cost of a more difficult formal proof.

In this section, we present the formal verification of three different implementations using respectively sorted lists, AVL trees and red-black trees. These three implementations are functors taking an ordered type `X` as argument.

4.1 Sorted lists

Sets implemented as sorted lists offer poor performances but there are at least two reasons to start with such an implementation. First, this is a quick way to debug our signature `S` and, when done, to show its logical consistency[5]. Second, some of the operations over lists are reused later in the code or verification of the other two implementations based on binary trees.

The verification is (almost) straightforward.

4.2 AVL trees

The next implementation to be verified is the `Set` module from OCAML standard library [2]. This is a heavily used library, including in OCAML's own code. It implements sets using AVL trees [4], that are binary search trees where the difference between the heights of any two sibling trees can not exceed a given value Δ. Although $\Delta = 1$ is an admissible choice [4], the OCAML implementation relaxes it to $\Delta = 2$, making a compromise between the overall balancing and the cost of rebalancing when inserting or deleting.

The COQ formalization implements signature `Sdep`, following OCAML code as close as possible. First a type for trees is introduced, where the height is stored for greater efficiency:

```
Inductive tree : Set :=
| Leaf : tree
| Node : tree → elt → tree → Z → tree.
```

The property of being a binary search tree, `bst`, is then defined as an inductive predicate:

```
Inductive bst : tree → Prop :=
| BSLeaf :
    (bst Leaf)
| BSNode : (x:elt)(l,r:tree)(h:Z)
    (bst l) → (bst r) →
    (lt_tree x l) → (gt_tree x r) →
    (bst (Node l x r h)).
```

[5] We indeed rephrased several specifications while doing this first verification.

where lt_tree x l (resp. gt_tree x l) states that any element in l is smaller (resp. greater) than x. Similarly, the balancing property is introduced as another inductive predicate:

```
Inductive avl : tree → Prop :=
| RBLeaf :
    (avl Leaf)
| RBNode : (x:elt)(l,r:tree)(h:Z)
    (avl l) → (avl r) →
    `-2 <= (height l) - (height r) <= 2` →
    `h = (max (height l) (height r)) + 1` →
    (avl (Node l x r h)).
```

Finally, the type t for sets is a record containing a tree and proofs that it is a well-balanced binary search tree:

```
Record t : Set := t_intro {
  the_tree :> tree;
  is_bst : (bst the_tree);
  is_avl : (avl the_tree)
}.
```

Properties bst and avl could have been defined simultaneously but separating them eases the proofs since most of the time one of the two is not relevant for the property to be proved.

Verification. Verifying the operations is mostly a matter of finding the precise specifications, where the OCAML code is only providing a few laconic comments. For instance, one of the internal function (bal l x r) is informally specified as

> "Same as create, but performs one step of rebalancing if necessary. Assumes l and r balanced."

but its precise specification is (among other things):

> "Assumes $|(\texttt{height l}) - (\texttt{height r})| \leq 3$. The size of the returned tree is either $\max(\texttt{height l}, \texttt{height r})$ or $\max(\texttt{height l}, \texttt{height r}) + 1$, and is always the latter when $|(\texttt{height l}) - (\texttt{height r})| \leq 2$.

Looking for these specifications, we actually discovered balancing bugs in OCAML code: two internal functions were building incorrectly balanced trees while they were supposed to. (The sets which were built were correct, though, i.e. were containing the right elements.) Patches have been quickly provided by the OCAML team and we could verify the new code without trouble.

A termination challenge. Only one operation poses a real verification challenge: the `compare` function providing a total ordering over sets. The idea is quite simple. Comparing two sets is just a matter of comparing the sorted lists of their elements in a lexicographic way. But the algorithm used is tricky. Instead of first building the two lists, the code is building them *lazily*, as soon as elements are needed for the comparison, using a technique of *deforestation* [15]. The problem is generalized to the comparison of two lists of trees, done as follows:

```
let rec compare_aux l1 l2 = match (l1, l2) with
  | ([], []) → 0
  | ([], _)  → -1
  | (_, []) → 1
  | (Empty :: t1, Empty :: t2) →
      compare_aux t1 t2
  | (Node(Empty, v1, r1, _) :: t1, Node(Empty, v2, r2, _) :: t2) →
      let c = Ord.compare v1 v2 in
      if c <> 0 then c else compare_aux (r1::t1) (r2::t2)
  | (Node(l1, v1, r1, _) :: t1, t2) →
      compare_aux (l1 :: Node(Empty, v1, r1, 0) :: t1) t2
  | (t1, Node(l2, v2, r2, _) :: t2) →
      compare_aux t1 (l2 :: Node(Empty, v2, r2, 0) :: t2)
```

Proving the termination of this function is hard. Indeed, the last two cases may recursively call `compare_aux` with “bigger” arguments when `l1` (resp. `l2`) is `Empty`. The reason why it terminates involves a global argument: the first elements of the lists will eventually become both `Empty` and will then fall into the fourth case of the pattern matching. Fortunately, the code can be slightly changed to recover a simpler termination argument, at the extra cost of two additional cases but without any loss of efficiency. This modified version is proved correct.

4.3 Red-black trees

Red-black trees [8] are another kind of balanced binary search trees. Nodes are colored either red or black and any red-black tree must satisfy the following two invariants:

- A red node has no red child;
- Every path from the root to a leaf contains the same number of black nodes.

Okasaki nicely introduces red-black trees in a functional setting [14] but only the membership and insertion operations are given. Xi specifies this code in Dependent ML [16] but it is also restricted to the insertion operation. Even Adams general approach balanced binary search trees [3] does not apply nicely to red-black trees. Generally speaking, we could not find a comprehensive implementation of finite sets using red-black trees and we wrote our own. This code is available from the web site of the formalization.

As for AVL trees, the Coq formalization implements signature `Sdep` following Ocaml code as close as possible. First, colored trees are defined:

```
Inductive color : Set := red : color | black : color.

Inductive tree : Set :=
| Leaf : tree
| Node : color → tree → elt → tree → tree.
```

The binary search tree property bst is similar to the one for AVL trees. Then the red-black trees invariant is introduced as an inductive predicate rbtree parameterized by the height of black nodes:

```
Inductive rbtree : nat → tree → Prop :=
| RBLeaf :
    (rbtree O Leaf)
| RBRed : (x:elt)(l,r:tree)(n:nat)
    (rbtree n l) → (rbtree n r) →
    (is_not_red l) → (is_not_red r) →
    (rbtree n (Node red l x r))
| RBBlack : (x:elt)(l,r:tree)(n:nat)
    (rbtree n l) → (rbtree n r) →
    (rbtree (S n) (Node black l x r)).
```

where is_not_red has an obvious meaning. Finally, everything is collected together in a record type:

```
Record t : Set := t_intro {
  the_tree :> tree;
  is_bst : (bst the_tree);
  is_rbtree : (EX n:nat | (rbtree n the_tree))
}.
```

Again, the formalization is roughly a matter of finding the right specification for each function, followed by a quite long process of Coq tactics scripting (figures are given in the conclusion). Some proofs from the AVL trees formalization could be reused with slight modifications (e.g. the compare operation).

5 Conclusion

In this article, we presented the full formalization in Coq of three applicative implementations of finite sets libraries, including AVL and red-black trees. To our knowledge, this is the first formal proof of a full set of operations over these two kinds of balanced binary search trees.

This article also demonstrates the benefits of modules and functors in a logical framework, and their relevance for program proving. More precisely, the adequacy between OCaml and Coq module systems allows the formalization of significant pieces of code. Correct-by-construction functorized code can be obtained using Coq extraction, which is a real improvement.

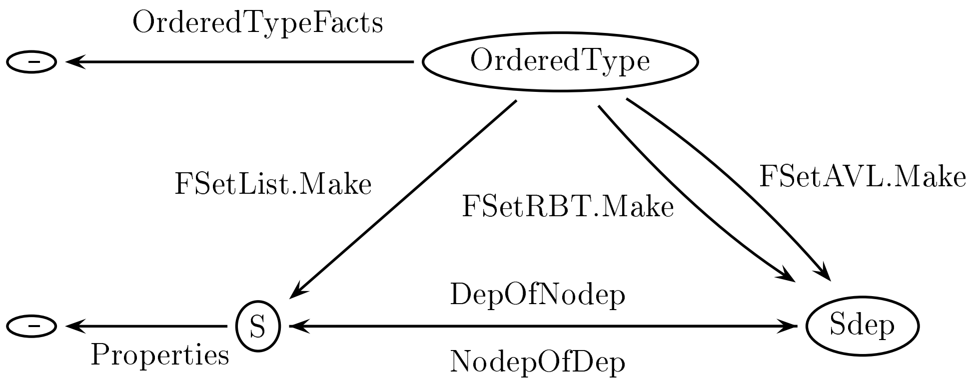

Figure 3: The five signatures and the seven functors

Overall picture. Figure 3 summarizes the dependencies between the five signatures and the seven functors used in this formalization. The following table details the size of the formalization, in terms of the size of code proved correct (not meaningful for sorted lists, which were implemented directly in Coq) and size of the Coq development. The formalization roughly amounts to 2 men-month.

	specs	lists	AVL	RBT	**total**
lines of code		114	231	314	545
lines of Coq specs	1160	375	532	405	2472
lines of Coq proofs	1900	537	1800	1280	5517

Extraction and benchmark. Once the formalization done, Ocaml code can be automatically extracted from the proofs [12, 13]. Thus it can be compared to the original code. We run a little benchmark comparing Ocaml's `Set` module (AVL), the extraction of its formalization (ε-AVL), a manual implementation of red-black trees (RBT) and the extraction of its formalization (ε-RBT). The benchmark consists in testing operations on randomly generated sets of various sizes[6]. Results are shown Figure 4.

The timings are very close, apart from ε-AVL when trees are built, that is for all operations except `mem`. The reason is that arithmetical computations over heights are done using Coq arbitrary precision arithmetic extracted to Ocaml, which can not compete with the hardware arithmetic used in Ocaml `Set`. We could parameterize the whole formalization of AVL trees with respect to the arithmetic used for computing heights, using yet another functor. But we would loose the benefits of the `Omega` tactic (the decision procedure for Presburger arithmetic) which is of heavy use in this development. A more realistic workaround would be an automatic substitution of hardware arithmetic for Coq arithmetic at extraction time, but this is not yet a Coq feature.

Acknowledgements. We are grateful to Xavier Leroy for suggesting the verification of Ocaml's AVL trees and for having provided patches almost immediately. We also thank

[6]The benchmark sources can be obtained from the authors.

	AVL	ε-AVL	RBT	ε-RBT
add-1	3.01	13.80	3.38	4.02
add-2	15.50	53.20	13.40	16.70
add-3	5.40	22.30	4.57	6.55
mem-1	9.52	10.20	11.40	12.10
mem-2	11.10	11.70	13.00	13.60
mem-3	8.31	8.83	9.01	9.47
mem-4	9.93	10.40	10.60	11.00
remove-1	4.67	18.70	5.92	6.85
remove-2	4.20	17.70	4.81	5.58
union	11.40	46.50	12.10	17.30
inter	4 .76	17.70	4.76	6.19
diff	4.68	17.70	4.71	5.76
subset	11.50	46.30	11.20	14.00
compare	11.50	45.70	11.70	13.90

Figure 4: Benchmark results (in seconds)

Benjamin Monate for the very nice user-interface CoqIDE and Diego Olivier Fernandez Pons for comments on implementing red-black trees.

References

[1] The Coq Proof Assistant. `http://coq.inria.fr/`.

[2] The Objective Caml language. `http://caml.inria.fr/`.

[3] Stephen Adams. Functional pearls: Efficient sets – a balancing act. *Journal of Functional Programming*, 3(4):553–561, October 1993. Expanded version available as Technical Report CSTR 92-10, University of Southampton.

[4] G. M. Adel'son-Vel'skiĭ and E. M. Landis. An algorithm for the organization of information. *Soviet Mathematics–Doklady*, 3(5):1259–1263, September 1962.

[5] Jacek Chrząszcz. Implementing modules in the system Coq. In *16th International Conference on Theorem Proving in Higher Order Logics*, University of Rome III, September 2003.

[6] Jacek Chrząszcz. *Modules in Type Theory with generative definitions*. PhD thesis, Warsaw University and Université Paris-Sud, 2003. To be defended.

[7] Judicaël Courant. A Module Calculus for Pure Type Systems. In *Typed Lambda Calculi and Applications 97*, Lecture Notes in Computer Science, pages 112 – 128. Springer-Verlag, 1997.

[8] Leo J. Guibas and Robert Sedgewick. A dichromatic framework for balanced trees. In *19th Annual Symposium on Foundations of Computer Science*, pages 8–21, Ann Arbor, Michigan, 16-18 October 1978. IEEE.

[9] Robert Harper and Mark Lillibridge. A type-theoretic approach to higher-order modules with sharing. In *Conference record of POPL '94: 21st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 123–137, Portland, OR, January 1994.

[10] Ralf Hinze. Constructing red-black trees. In editor Chris Okasaki, editor, *Proceedings of the Workshop on Algorithmic Aspects of Advanced Programming Languages, WAAAPL'99*, pages 89–99, Paris, France, September 1999. Also technical report of Columbia University, CUCS-023-99.

[11] Xavier Leroy. A modular module system. *Journal of Functional Programming*, 10(3):269–303, 2000.

[12] Pierre Letouzey. A New Extraction for Coq. In Herman Geuvers and Freek Wiedijk, editors, *Types for Proofs and Programs, Second International Workshop, TYPES 2002, Berg en Dal, The Netherlands, April 24-28, 2002*, volume 2646 of *Lecture Notes in Computer Science*. Springer-Verlag, 2003.

[13] Pierre Letouzey. *Programmation fonctionnelle certifiée en Coq*. PhD thesis, Université Paris Sud, 2003.

[14] Chris Okasaki. *Purely Functional Data Structures*. Cambridge University Press, 1998.

[15] Philip Wadler. Deforestation: transforming programs to eliminate trees. *Theoretical Computer Science*, 73:231–248, 1990.

[16] Hongwei Xi. Dependently Typed Data Structures. In *Proceedings of Workshop of Algorithmic Aspects of Advanced Programming Languages (WAAAPL '99)*, pages 17–32, Paris, September 1999.