Producing All Ideals of a Forest, Formally (Verification Pearl)

Jean-Christophe Filliâtre, Mário Pereira

Producing All Ideals of a Forest, Formally (Verification Pearl)

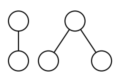
Jean-Christophe Filliâtre[1,2] and Mário Pereira[1,2]

[1] Lab. de Recherche en Informatique, Univ. Paris-Sud, CNRS, Orsay, F-91405
[2] INRIA Saclay – Île-de-France, Orsay, F-91893

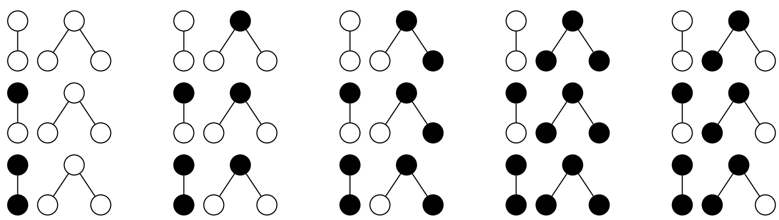
Abstract. In this paper we present the first formal proof of an implementation of Koda and Ruskey's algorithm, an algorithm for generating all ideals of a forest poset as a Gray code. One contribution of this work is to exhibit the invariants of this algorithm, which proved to be challenging. We implemented, specified, and proved this algorithm using the Why3 tool. This allowed us to employ a combination of several automated theorem provers to discharge most of the verification conditions, and the Coq proof assistant for the remaining two.

1 Introduction

Given a forest, we consider the problem of coloring its nodes in black and white, such that a white node only has white descendants. Consider for instance this forest:



It has exactly 15 colorings, which are the following:



Koda and Ruskey proposed a very nice algorithm [4] to generate all these colorings.[3] This is a Gray code algorithm, which only changes the color of one node to move from one coloring to the next one. If we read the figure above in a zig-zag way, we can notice that any coloring is indeed obtained from the previous one by changing the color of exactly one node.

This research was partly supported by the Portuguese Foundation for Sciences and Technology (grant FCT-SFRH/BD/99432/2014) and by the French National Research Organization (project VOCAL ANR-15-CE25-008).

[3] Such a coloring has a mathematical interpretation as an ideal of a forest poset.

There are many ways to implement Koda-Ruskey's algorithm. Koda and Ruskey themselves give two implementations in their paper. Filliâtre and Pottier propose several implementations based on higher-order functions and their defunctionalization [1]. Knuth has two implementations in C, including one using coroutines [3]. In particular, Knuth makes the following comment:

> [...] I think it's a worthwhile challenge for people who study the science of computer programming to verify that these two implementations both define the same sequence of bitstrings.

Before trying to verify Knuth's intricate C code, a reasonable first step is to work out the invariants of Koda-Ruskey's algorithm on a simpler implementation. This is what we do in this paper, using the Why3 system. To our knowledge, this is the first formal proof of this algorithm.

This paper is organized as follows. Sec. 2 describes our implementation in Why3. Then Sec. 3 goes over the formal specification. Finally, Sec. 4 details the most interesting parts of the proof. The Why3 source code and its proof can be found at `http://toccata.lri.fr/gallery/koda_ruskey.en.html`.

2 Implementation

Our implementation of Koda-Ruskey's algorithm is given in Fig. 1. The syntax of Why3 is close to that of OCaml, and we explain it whenever necessary. The algebraic datatype of forests is declared on lines 1–3. A forest is either empty (constructor `E`) or composed of an integer node together with two forests, namely the forest of its children nodes and its sibling forest (constructor `N`). One can notice that the type `forest` is isomorphic to a list of pairs of nodes and forests.

The type of colors is introduced on line 4. The entry point is function `main` (lines 24–25). It takes an array `bits` as argument, to hold the coloring, and a forest `f0`. It then calls a recursive function `enum`, which implements the core of the algorithm.

Function `enum` operates over a stack of forests, using the predefined type `list` of Why3 (with constructors `Nil` and `Cons`). On entry, function `enum` inspects the stack. It will never be empty (line 9). If the stack is reduced to a single empty forest, we have just discovered a new coloring. We are free to do whatever we want with the contents of array `bits` (line 10), such as printing it, storing it, etc. If the stack starts with an empty forest, we skip it (line 11). Otherwise, the top of the stack contains a non-empty tree, with a root node `i`, a children forest `f1`, and a sibling forest `f2` (line 12). If node `i` is white (line 13), we first enumerate the colorings of `f2` together with the remaining `st'` of the stack (line 14), then we blacken node `i` (line 15), and finally we enumerate the colorings of `f1`, interleaving them with the colorings of `f2` and `st'`. If node `i` is black (line 17), the process is reversed. First, we enumerate the colorings of `f1` (line 18), so that all nodes of `f1` are white again at the end. Then we whiten node `i` (line 19). Finally, we enumerate the colorings of `f2` (line 20).

```
type forest =
  | E
  | N int forest forest

type color = White | Black

let rec enum (bits: array color) (st: list forest)  =
  match st with
  | Nil → absurd
  | Cons E Nil → ... (* visit array bits *) ...
  | Cons E st' → enum bits st'
  | Cons (N i f1 f2) st' →
      if bits[i] = White then begin
        enum bits (Cons f2 st');
        bits[i] ← Black;
        enum bits (Cons f1 (Cons f2 st'))
      end else begin
        enum bits (Cons f1 (Cons f2 st'));
        bits[i] ← White;
        enum bits (Cons f2 st')
      end
  end

let main (bits: array color) (f0: forest) =
  enum bits (Cons f0 Nil)
```

Fig. 1: An implementation of Koda-Ruskey's algorithm.

3 Specification

In this section we give function `main` a specification. The specification of function `enum` is considered being part of the proof and thus only described in the next section. The first requirement over function `main` is to have array `bits` large enough to hold all the nodes of the forest. So we start by defining the number of elements in a forest:

```
function size_forest (f: forest) : int = match f with
  | E → 0
  | N _ f1 f2 → 1 + size_forest f1 + size_forest f2
  end
```

In Why3, the `function` keyword introduces a logical function, *i.e.*, a function with no side-effects and whose termination is checked automatically, and that one can use in a specification context. We use `size_forest` to introduce the first precondition of function `main`:

```
let main (bits: array color) (f0: forest)
  requires { size_forest f0 = length bits } ...
```

To execute correctly, the program also requires the forest to have nodes numbered with distinct integers that are also valid indexes in array `bits`. These conditions are expressed, respectively, by predicates `no_repeated_forest` and `between_range_forest`, as follows:

```
predicate no_repeated_forest (f: forest) = match f with
  | E → true
  | N i f1 f2 →
      no_repeated_forest f1 && no_repeated_forest f2 &&
      not (mem_forest i f1) && not (mem_forest i f2) &&
      disjoint f1 f2
  end

predicate between_range_forest (i j: int) (f: forest) =
  forall n. mem_forest n f → i ≤ n < j
```

where `mem_forest` expresses that an element belongs to a forest:

```
predicate mem_forest (n: int) (f: forest) = match f with
  | E → false
  | N i f1 f2 → i = n || mem_forest n f1 || mem_forest n f2
  end
```

and `disjoint` indicates that two trees have disjoint sets of nodes:

```
predicate disjoint (f1 f2: forest) =
  forall x. mem_forest x f1 → mem_forest x f2 → false
```

To write more succinct specifications in the following, we combine predicates `between_range_forest` and `no_repeated_forest` into a single predicate `valid_nums_forest`, which is added to the precondition of `main`.

```
predicate valid_nums_forest (f: forest) (n: int) =
  between_range_forest 0 n f && no_repeated_forest f

let main (bits: array color) (f0: forest)
  requires { valid_nums_forest f0 (size_forest f0) } ...
```

We now turn to the part of the specification related to the enumeration of colorings. A coloring is a map from nodes, which are integers, to values of type `color`:

```
  type coloring = map int color
```

At the beginning of the algorithm, all nodes of the forest must be colored white. We introduce a predicate `white_forest` to say so.

```
predicate white_forest (f: forest) (c: coloring) = match f with
  | E → true
  | N i f1 f2 → c[i] = White && white_forest f1 c && white_forest f2 c
end
```

This predicate traverses the forest and checks that for each node `i`, its color `c[i]` is `White`. As for functions, termination of recursive predicates is automatically also checked. We can now use this predicate in the precondition of `main`:

```
let main (bits: array color) (f0: forest)
  requires { white_forest f0 bits.elts } ...
```

Here `bits.elts` is the map modeling the contents of array `bits`, which happens to have type `coloring`.

Upon termination, the program must have enumerated all colorings, each coloring being visited exactly once. Since the code is not storing the colorings, we extend it with *ghost* code to do that. A ghost reference, `visited`, is declared to hold the sequence of colorings enumerated so far:

```
val ghost visited: ref (seq coloring)
```

(Sequences are predefined in Why3 standard library.) The idea is that this reference is updated each time a new coloring is found, on line 10 of the program in Fig. 1.

To express that `main` enumerates all colorings exactly once, we specify that all colorings in `visited` are valid and pairwise distinct colorings, and that there are the expected number of colorings. The latter is easily defined recursively:

```
function count_forest (f: forest) : int = match f with
  | E         → 1
  | N _ f1 f2 → (1 + count_forest f1) * count_forest f2
  end
```

Indeed, an empty forest has exactly one coloring (the empty coloring), and colorings of a non-empty forest are obtained by combining any coloring for the first tree with any coloring for the remaining forest. Last, the coloring of a tree is either all white (hence 1) or a black root with any coloring of the children forest. The postcondition of `main` states that we have enumerated this number of colorings:

```
let main (bits: array color) (f0: forest)
  ensures  { length !visited = count_forest f0 } ...
```

To be valid, a coloring must respect the constraint that if a node is colored white then its children forest must be all white. The predicate `valid_coloring` checks this constraint:

```
predicate valid_coloring (f: forest) (c: coloring) =
  match f with
  | E → true
  | N i f1 f2 →
      valid_coloring f2 c &&
      match c[i] with
      | White → white_forest f1 c
      | Black → valid_coloring f1 c
      end
  end
```

Each time a white node is reached, we use predicate `white_forest` to ensure that its children forest is white.

```
let rec enum (bits: array color) (ghost f0: forest) (st: list forest)
  requires { st ≠ Nil }
  requires { size_forest f0 = length bits }
  requires { valid_nums_forest f0 (length bits) }
  requires { sub st f0 bits.elts }
  requires { any_stack st bits.elts }
  requires { valid_coloring f0 bits.elts }
  ensures  { forall i. not (mem_stack i st) → bits[i] = (old bits)[i] }
  ensures  { inverse st (old bits).elts bits.elts }
  ensures  { valid_coloring f0 bits.elts }
  ensures  { stored_solutions f0 bits.elts st (old !visited) !visited }
  variant  { size_stack st, st }
```

Fig. 2: Specification of function `enum`.

Comparing two colorings requires to ignore values outside of the array range. Thus we introduce predicate `eq_coloring` to state that two colorings coincide on a given range $0..n-1$:

```
predicate eq_coloring (n: int) (c1 c2: coloring) =
  forall i. 0 ≤ i < n → c1[i] = c2[i]
```

We are now in position to give the full code and specification of function `main`:

```
let main (bits: array color) (f0: forest)
  requires { size_forest f0 = length bits }
  requires { valid_nums_forest f0 (size_forest f0) }
  requires { white_forest f0 bits.elts }
  ensures  { length !visited = count_forest f0 }
  ensures  { let n = length !visited in
             forall j. 0 ≤ j < n →
             valid_coloring f0 !visited[j] &&
             forall k. 0 ≤ k < n → j ≠ k →
               not (eq_coloring (length bits) !visited[j] !visited[k]) }
= visited := empty;
  enum bits f0 (Cons f0 Nil)
```

Note that `main` assigns `visited` to the `empty` sequence before calling `enum`. The forest `f0` is also passed to `enum` as an extra, ghost argument.

4 Proof

As shown in Fig. 1, program `main` simply amounts to a call to `enum`. So, in order to prove that `main` respects its specification we need to specify and prove correct function `enum`. In this section we go over the most subtle points in the specification and proof of `enum`. The complete specification for this function is shown in Fig. 2.

Function `enum` operates on a stack of forests, and we need to relate that stack to the original forest `f0` (which is passed to `enum` as a ghost argument). To do so, we introduce a predicate $\texttt{sub}\ st\ f\ c$ that relates a stack st, a forest f, and a coloring c. It is defined with the following inference rules:

$$\frac{}{\texttt{sub}\ [f]\ f\ c} \qquad \frac{\texttt{sub}\ st\ f_2\ c}{\texttt{sub}\ st\ (\texttt{N}\ i\ f_1\ f_2)\ c} \qquad \frac{\texttt{sub}\ st\ f_1\ c \quad c[i] = \texttt{Black}}{\texttt{sub}\ (st \mathbin{++} [f_2])\ (\texttt{N}\ i\ f_1\ f_2)\ c}$$

The first rule states that a stack containing a single forest f is a sub-forest of f itself. ($[f]$ is a notation for a one-element list.) The second rule states that we can skip the left tree (i, f_1) of a forest $(\texttt{N}\ i\ f_1\ f_2)$. The third rule states that we can plunge into f_1 provided $c[i]$ is black and f_2 appears at the end of the stack. (Operator `++` is list concatenation). In Why3, such a set of inference rules is defined as an inductive predicate:

```
inductive sub stack forest coloring =
| Sub_reflex:
    forall f, c. sub (Cons f Nil) f c
| Sub_brother:
    forall st i f1 f2 c.
    sub st f2 c → sub st (N i f1 f2) c
| Sub_append:
    forall st i f1 f2 c.
    sub st f1 c → c[i] = Black →
    sub (st ++ Cons f2 Nil) (N i f1 f2) c
```

We use this predicate in `enum`'s precondition, with the current stack, the initial forest `f0`, and the current coloring (line 5). Together with preconditions in lines 2–4, we are already in position to prove safety of function `enum`. Indeed, nodes found in the stack do belong to `f0`, according to `sub`, and thus are legal array indices.

To specify what `enum` does, we need to characterize the final coloring in the enumeration (*e.g.*, the bottom right coloring in the 15 colorings on page 1). Indeed, for the algorithm to work, it has to enumerate all colorings in a reverse order when called on such a final coloring, ending on a white forest. Since the algorithm is interleaving the colorings for the various trees of the forest, the final configuration depends on the parity of these numbers of colorings. So we first introduce a predicate `even_forest f` which means that forest `f` has an even number of colorings:

```
predicate even_forest (f: forest) = match f with
  | E          → false
  | N _ f1 f2 → not (even_forest f1) || even_forest f2
  end
```

Though we could define it instead as `count_forest` being even, we prefer this direct definition, which saves us some arithmetical reasoning. We can now define what is the final coloring of a forest:

```
predicate final_forest (f: forest) (c: coloring) = match f with
```

```
  | E → true
  | N i f1 f2 →
      c[i] = Black && final_forest f1 c &&
      if not (even_forest f1) then white_forest f2 c
      else final_forest f2 c
  end
```

Though we can see `final_forest` as the dual of `white_forest`, from the algorithm point of view, it is clear that a final forest is not a black forest (as one can see on page 1). Function `enum` requires all forests in the stack to be either white or final. To say so, we introduce the following recursive predicate:

```
predicate any_stack (st: stack) (c: coloring) = match st with
  | Nil → true
  | Cons f st →
      (white_forest f c || final_forest f c) && any_stack st c
  end
```

It appears as a precondition on line 6.

From a big-step perspective, Koda-Ruskey's algorithm is switching from a white coloring to a final coloring and conversely. But `enum` is operating over a stack of forests and thus requires us to be more precise. For the tree on top of the stack, we are indeed switching states. However, for the next tree (its right sibling in the same forest, if any, or the next tree in the stack, otherwise), the state changes only if the first tree has an odd number of colorings. Otherwise, it is kept unchanged. To account for this inversion, we introduce the following predicate that relates a stack `st` and two colorings, namely the first coloring `c1` and the last coloring `c2`:

```
predicate inverse (st: stack) (c1 c2: coloring) =
  match st with
  | Nil → true
  | Cons f st' →
      (white_forest f c1 && final_forest f c2 ||
       final_forest f c1 && white_forest f c2) &&
      if even_forest f then
        unchanged st' c1 c2
      else
        inverse st' c1 c2
  end
```

Note that the coloring of the first forest in the stack is always inverted, while the inversion of the remaining of the stack depends on the parity of the first forest. The predicate `unchanged st' c1 c2` states that `c1` and `c2` coincide on any node in the stack `st'`. The postcondition on line 9 in Fig. 2 relates the initial contents of array `bits` (written `old bits`) to its final contents using predicate `inverse`.

We briefly go over the remaining clauses in the specification of `enum`. The stack is never empty (line 2). The initial forest `f0` has as many elements as the `bits` array (line 3) and is correctly numbered from 0 (line 4). In both pre- and post-state, the coloring must be valid w.r.t. `f0` (lines 7 and 10). A frame postcondition ensures that any element outside of the stack is left unchanged (line 8). We characterize the sequence of enumerated colorings with a predicate `stored_solutions` (line 11), not shown here. It means that `visited` has been augmented with new, valid, and pairwise distinct colorings, which coincide with array `bits` outside of the stack nodes. Finally, we ensure termination with a lexicographic variant (line 12). In all cases but one, the size of the stack is decreasing, when defined as its total number of nodes, as follows:

```
function size_stack (st: stack) : int = match st with
  | Nil → 0
  | Cons f st → size_forest f + size_stack st
  end
```

The last case is when the stack is of the form `Cons E st'`, for which we perform a recursive call on `st'`. The number of nodes remains the same, but the stack is structurally smaller, hence the lexicographic variant.

Proof Statistics. To make the proof of `enum` and `main` fully automatic, we introduce 19 proof hints in the body of `enum` and 37 auxiliary lemmas. Many of these lemmas require a proof by induction, which is done in Why3 by first applying a dedicated transformation (interactively, from the Why3 IDE) and then calling automated theorem provers. The table below summarizes the number of VCs and the verification time.

	number of VCs	automatically proved	verification time
lemmas	102	100 (98%)	14.72 s
`enum`	94	94 (100%)	47.51 s
`main`	7	7 (100%)	0.07 s
total	203	201 (99%)	62.30 s

Two VCs are proved interactively using Coq. These proofs amount to 55 lines of Coq tactics, including the `why3` tactic that allows to automatically discharge some Coq sub-goals using SMT solvers. All other VCs are proved automatically, using a combination of theorem provers as follows:

prover	VCs proved
Alt-Ergo 1.01	139
CVC4 1.4	57
Z3 4.4.0	3
CVC3 2.4.1	1
Eprover 1.8-001	1

Our proof process consists in calling Alt-Ergo first. When it does not succeed, we switch to CVC4. And so on. So the numbers above should not be interpreted as "Alt-Ergo discharges 139 VCs and CVC4 only 57". Though we could call all provers on all VCs, we choose not to do this in practice to save time. A more detailed table is available on-line at `http://toccata.lri.fr/gallery/koda_ruskey.en.html`.

5 Conclusion

In this paper we presented a formal verification of an implementation of Koda-Ruskey's algorithm using Why3. To our knowledge, this is the first formal proof of this algorithm. The main contribution of this paper is the definition of the algorithm's invariants (mostly, the definition of predicates `any_stack` and `inverse`). We argue that such definitions could be readily reused in other proofs of this algorithm, whatever the choice of implementation and of verification tool (*e.g.*, Dafny [5], VeriFast [2], or Viper [6]).

We intend to improve our verification with a proof that `count_forest` is indeed the right number of colorings. One way to do that would be to implement a naive enumeration of all colorings, with an obvious soundness proof. We are also interested in verifying higher-order implementations of Koda-Ruskey's algorithm, such as the ones by Filliâtre and Pottier [1]. This means extending Why3 with support for effectful higher-order functions.

Acknowledgments. We thank Claude Marché for his comments on earlier versions of this paper.

References

1. Filliâtre, J.C., Pottier, F.: Producing All Ideals of a Forest, Functionally. Journal of Functional Programming 13(5), 945–956 (September 2003)
2. Jacobs, B., Smans, J., Philippaerts, P., Vogels, F., Penninckx, W., Piessens, F.: VeriFast: A powerful, sound, predictable, fast verifier for C and Java. In: Bobaru, M.G., Havelund, K., Holzmann, G.J., Joshi, R. (eds.) NASA Formal Methods. Lecture Notes in Computer Science, vol. 6617, pp. 41–55. Springer (2011)
3. Knuth, D.E.: An implementation of Koda and Ruskey's algorithm (June 2001), `http://www-cs-staff.stanford.edu/~knuth/programs.html`
4. Koda, Y., Ruskey, F.: A Gray Code for the Ideals of a Forest Poset. Journal of Algorithms (15), 324–340 (1993)
5. Leino, K.R.M.: Dafny: An automatic program verifier for functional correctness. In: LPAR-16. Lecture Notes in Computer Science, vol. 6355, pp. 348–370. Springer (2010)
6. Müller, P., Schwerhoff, M., Summers, A.J.: Viper: A verification infrastructure for permission-based reasoning. In: Jobstmann, B., Leino, K.R.M. (eds.) Verification, Model Checking, and Abstract Interpretation (VMCAI). LNCS, vol. 9583, pp. 41–62. Springer-Verlag (2016)