JFP **13** (5): 945–956, September 2003. © 2003 Cambridge University Press
DOI: 10.1017/S0956796803004763 Printed in the United Kingdom

Producing all ideals of a forest, functionally

JEAN-CHRISTOPHE FILLIÂTRE
Laboratoire de Recherche en Informatique, Université Paris Sud, 91405 Orsay Cedex, France
(*e-mail:* `Jean-Christophe.Filliatre@lri.fr`)

FRANÇOIS POTTIER
INRIA Rocquencourt, B.P. 105, 78153 Le Chesnay Cedex, France
(*e-mail:* `Francois.Pottier@inria.fr`)

Abstract

We present functional implementations of Koda and Ruskey's algorithm for generating all ideals of a forest poset as a Gray code. Using a continuation-based approach, we give an extremely concise formulation of the algorithm's core. Then, in a number of steps, we derive a first-order version whose efficiency is comparable to that of a C implementation given by Knuth.

1 Introduction

It is sometimes said that functional programming languages are inherently less efficient than their imperative counterparts. Today, such an opinion has become a stereotype without substance. Yet, we still confront it regularly, and must provide convincing "practical" evidence. In this paper, we show how a complex algorithm, heretofore presented only in an imperative form, can be expressed in a programming language equipped with first-class functions. We obtain code that is more concise, significantly easier to prove correct, yet equally efficient as the original. Then, we derive a first-order version of our code, which can be easily implemented in C, if desired.

The algorithm we are interested in is due to Koda & Ruskey (1993). It enumerates the ideals of certain finite partially ordered sets – namely, those whose Hasse diagram is a forest – as a Gray code. In general, a Gray code is a sequence of words such that two consecutive words differ by only one letter. A widely studied particular case consists in enumerating all binary integers, from $00\cdots0$ to $11\cdots1$, as a Gray code. Gray codes find application in mathematics, electrical engineering, optics, scheduling, network reliability, etc. In fact, a whole section is devoted to them in the fourth volume of Knuth's *Art of Computer Programming*. A preliminary version of this section is currently available electronically (Knuth, 2001b). While writing it, Knuth took interest in Koda and Ruskey's algorithm, and published two implementations of it (Knuth, 2001a). Our interest arose from these readings.

Koda and Ruskey's algorithm can be described in a simple way. The task is to enumerate all *colorings* of a given, arbitrary forest. A coloring consists in marking

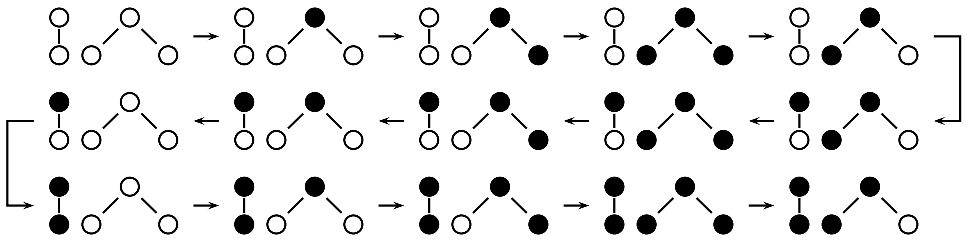
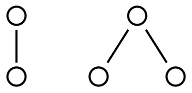


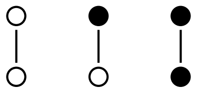
Fig. 1. Koda and Ruskey's algorithm applied to the forest (1).

every node as either black or white, with the sole constraint that all descendants of a white node be white as well. For instance, the following forest:

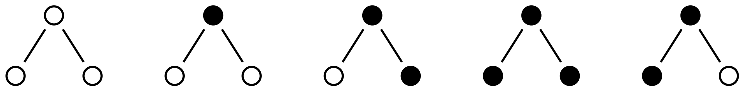

(1)

admits exactly 15 distinct colorings, all of which are given in Figure 1. By definition, a sequence of colorings forms a Gray code if and only if every coloring of the forest appears exactly once in it and two consecutive colorings differ by the color of exactly one node.

Let us illustrate the algorithm's functioning on the forest (1). The main idea is to interleave the sequences of colorings which correspond to each of the trees that form the forest. Here, one must interlace the sequence of the three colorings of the left-hand tree, namely:


(2)

with the sequence of the five colorings of the right-hand tree, given below:


(3)

Thus, the first line of Figure 1 exhibits the first coloring of the left-hand tree, combined successively with all colorings of the right-hand tree. The second line shows the second coloring of the left-hand tree, again combined with all colorings of the right-hand tree, but this time in reverse order – indeed, it is clear that the mirror image of a Gray code remains a Gray code. Lastly, the third line exhibits the third coloring of the left-hand tree and all colorings of the right-hand tree, this time again in their initial order.

There remains to explain how to enumerate all colorings of a tree. Let the first coloring be uniformly white. Then, to obtain the remainder of the sequence, color the root node black and enumerate all colorings of the forest formed by its children. The sequence thus obtained is indeed a Gray code, because (i) the first and second colorings differ only by the color of the root node, and (ii) from then on, the root node remains unaffected, and the sequence of the colorings of the children forms a Gray code by construction. This process is illustrated by (2) and (3) above. Note that the coloring where every node is black does not necessarily appear last in a sequence.

Koda & Ruskey's (1993) paper describes two versions of this algorithm, written as imperative pseudo-code and as Pascal code. One has complexity $O(nN)$, where n is the number of nodes in the forest and N is the number of its colorings, that is, the length of the Gray code to be produced. The other is a refinement with optimal complexity, namely $O(N)$. More recently, two C implementations were given by Knuth (Knuth, 2001a). All of these implementations are complex: they are typically 50–80 lines long and involve imperative modifications of subtle data structures.

The present paper describes an alternative approach to implementing Koda and Ruskey's algorithm. We begin with a simple algorithm (Section 2), which we first implement in a purely functional manner and then translate into a slightly more imperative style. Indeed, our programming language is Objective Caml (Leroy *et al.*, 2002), so it is natural to exploit – to some degree – its imperative features. However, it would be possible to use any language that supports first-class functions and mutable arrays, such as other ML dialects, Haskell, Lisp, Scheme, etc. In Section 3, we slightly modify the algorithm so as to achieve optimal complexity $O(N)$. Then, Sections 4 and 5 present refined implementations of the second algorithm, eliminating first-class functions in favor of lower-level representations, while preserving most of the simplicity afforded by our approach. Lastly, Section 6 compares our implementations with those proposed by Knuth, performance-wise.

2 A continuation-based algorithm

We represent a forest as a value of OCaml type `forest`, defined as follows:

```
type tree   = Node of int × forest
and forest = tree list
```

α `list` is OCaml's predefined type for lists of elements of type α. The list containing $x_1, x_2, \ldots, x_n$ in this order is written `[`x_1`;` x_2`; ...;` x_n`]`. The empty list is written `[]`. The addition of an element x at the beginning of a list l is written $x :: l$. The n nodes of the forest are labeled by the integers $0, 1, \ldots, n-1$ in an arbitrary manner.

The algorithm needs to maintain a current coloring. It also needs to display every coloring after it is computed. Thus, our purely functional implementation uses a combined I/O and state monad, whose OCaml signature is given in the top half of Figure 2. A state contains both the coloring, represented as an array of integers where 0 stands for white and 1 stands for black, and the output displayed so far, represented as a string. A computation is a state transformer, that is, a function from states to states. The state `create` n is the algorithm's initial state, where every node is colored white. The computation `update` i c colors node i with color c. The operation `get` i returns the color of node i. Finally, the computation `print` appends the description of the current coloring to the output string. Implementing this monad in OCaml is straightforward; we omit the code. To sequence computations, it is convenient to introduce the following infix operation, which is nothing but function composition:

```
type state = int array × string
type computation = state → state

val create : int → state
val update : int → int → computation
val get : int → state → int
val print : computation

let rec enum_forest k f s = match f with
  | [] → k s
  | t :: f → enum_tree (enum_forest k f) t s
and enum_tree k (Node (i,f)) s =
  if get i s == 0 then
    (k ++ update i 1 ++ enum_forest k f) s
  else
    (enum_forest k f ++ update i 0 ++ k) s
```

Fig. 2. A continuation-based version of Koda and Ruskey's algorithm (C0).

```
val (++) : computation → computation → computation
```

Let us now describe the core of the algorithm. Because trees and forests are defined in a mutually inductive way, we naturally define two mutually recursive functions `enum_tree` and `enum_forest`, which enumerate the colorings of a tree and of a forest, respectively. The key idea is to give these functions an extra argument `k`, of type `computation`, which will be called after every coloring of the tree (resp. forest) is complete. The function `k` may be viewed as a continuation, and we call it so in the following. The idea is, if the function `k` enumerates the colorings of a certain forest `f0`, then the computation `enum_forest k f` enumerates the colorings of the forest `f @ f0` and `enum_tree k t` those of the forest `t :: f0`, where `@` denotes forest concatenation.

The code is given in Figure 2; we refer to it as C0. Throughout, the variable `s` denotes the current state. Let us begin with `enum_forest`. If the forest is empty, we simply call the continuation. If, on the other hand, the forest contains at least one tree `t` next to a sub-forest `f`, then we enumerate the colorings of `t`, by applying `enum_tree` to `t`, with a new continuation that enumerates the colorings of `f` with continuation `k`. Let us now turn to `enum_tree`. Its task is slightly more complex, because it must enumerate the colorings either in one direction, or in the other, depending upon the current state. To determine which, `enum_tree` looks up the color of the tree's root, that is, `get i s`. If it is currently white, then the whole tree must be white. We have a complete coloring, so we signal the continuation `k`; then, we color the root black and enumerate its children's colorings using `enum_forest`. If, on the other hand, the root is currently black, we do the converse. That is, we first use `enum_forest` to enumerate the children's colorings in reverse order, which leaves all of the children entirely white; then, we color the root white, and signal the continuation `k`.

To run C0 on a forest `f`, one calls `enum_forest` with a continuation that displays the current coloring every time it is invoked, that is, `print`:

```
type computation = unit → unit

let rec enum_forest k = function
  | [] → k ()
  | t :: f → enum_tree (fun () → enum_forest k f) t
and enum_tree k (Node (i,f)) =
  if bits.(i) = 0 then begin
    k (); bits.(i) ← 1; enum_forest k f
  end else begin
    enum_forest k f; bits.(i) ← 0; k ()
  end
```

Fig. 3. A slightly more imperative implementation (C1).

```
enum_forest print f
```

This computation is then applied to a suitable initial state, namely `create n`, where `n` is the size of the forest `f`.

A slightly more imperative implementation. From here on, we use a native implementation of the monad described above, so as to obtain more idiomatic OCaml code. That is, the current coloring is now stored in a global array `bits`, while colorings are displayed by calling OCaml's standard library functions. As a result, computations operate only by side effect. The code is given in Figure 3; we refer to it as C1. The differences with respect to C0 are minor. The state parameter `s` disappears or is replaced with the `()` constant. The composition operator `++` is replaced with OCaml's native sequencing construct `;`. The current coloring is looked up and modified by reading and writing the global array `bits`. To run C1 on a forest `f`, one calls `enum_forest` with a continuation that displays the current contents of the array `bits` at every invocation:

```
enum_forest (fun () → (* display current configuration *)) f
```

Complexity. To assess C1's complexity, let us first introduce the two quantities in terms of which it is expressed, namely the forest's size and number of colorings. In the following, we use OCaml's list syntax for forests. We write `Node` f for a tree whose children form a forest f (and whose index is irrelevant). The size of a forest f (resp. of a tree t), written $n(f)$ (resp. $n(t)$), is the number of its nodes. It is defined inductively on the structure of trees and forests:

$$\begin{aligned} n(\texttt{[]}) &= 0 \\ n(t :: f) &= n(t) + n(f) \\ n(\texttt{Node}\ f) &= 1 + n(f) \end{aligned}$$

The number of colorings of a forest f (resp. of a tree t), written $N(f)$ (resp. $N(t)$), is defined similarly:

$$\begin{aligned} N(\texttt{[]}) &= 1 \\ N(t :: f) &= N(t) \times N(f) \\ N(\texttt{Node}\ f) &= 1 + N(f) \end{aligned}$$

Unless it is ambiguous, we write n and N for these two quantities. For the forest (1), we have $n = 5$ and $N = 15$.

We must make some assumptions about the cost of every operation. We ignore the cost of function calls: this slightly simplifies our computations, while affecting the final result only up to a constant factor. Two operations remain to be taken into account: modification of the `bits` array and closure construction. The former has constant cost; as for the latter, it is reasonable to assume a constant *amortized* cost. We consider both as unitary.

We write $F(k, f)$ for the total cost of applying `enum_forest` to a forest f with a continuation of cost k. Similarly, we write $T(k, t)$ for the cost of applying `enum_tree` to a tree t with a continuation of cost k. From the code C1, we derive the equations that govern these quantities:

$$F(k, \texttt{[]}) = k \tag{4}$$

$$F(k, t :: f) = 1 + T(F(k, f), t) \tag{5}$$

$$T(k, \texttt{Node } f) = 1 + k + F(k, f) \tag{6}$$

In equation (5), the unitary cost corresponds to closure construction. The closure itself is, by hypothesis, a continuation of cost $F(k, f)$, hence the second term. In equation (6), the unitary cost corresponds to updating the array. From these equations, it is easy to establish the following upper bounds:

$$\begin{aligned} F(k, f) &\leqslant N(f) \times (k + n(f)) \\ T(k, t) &\leqslant N(t) \times (k + n(t)) - 1 \end{aligned}$$

When one applies `enum_forest` to a forest f with a costless initial continuation, the upper bound simplifies to $N(f) \times n(f)$. Thus, we conclude that C1 has time complexity $O(nN)$. One may show, in a similar way, that the number of closures built during evaluation is bounded by $N(f) - 1$ and thus C1 has space complexity $O(N)$.

3 First refinement: pre-planning control

This time bound is not optimal; in fact, it is easy to see that C1 actually repeats some computations many times. Indeed, every time a given forest is traversed, the same continuation is built. In example (1), `enum_tree` is applied three times to the second tree; every time, it is passed a fresh continuation, whose effect is in fact the same (namely to call the initial continuation).

It is possible, with a slight modification to the algorithm, to factor out these repeated allocations. The idea is that `enum_tree` and `enum_forest`, instead of enumerating the colorings immediately, should now return a continuation (that is, a function of type `unit → unit`) that performs the enumeration when invoked. The modified code, which we refer to as C2, is given in Figure 4. It differs from C1 in three ways. First, when `enum_forest` is applied to an empty forest, it merely returns its continuation `k`, instead of executing it immediately. Second, when it is applied to a non-empty forest, it immediately invokes `enum_forest k f`, which

```
let rec enum_forest k = function
  | [] → k
  | t :: f → enum_tree (enum_forest k f) t
and enum_tree k (Node (i,f)) =
  let lf = enum_forest k f in
  fun () →
    if bits.(i) = 0 then begin
      k (); bits.(i) ← 1; lf ()
    end else begin
      lf (); bits.(i) ← 0; k ()
    end
```

Fig. 4. First refinement (C2).

returns a continuation; the need for an explicit delay (that is, a λ-abstraction) has been removed. Lastly, and most importantly, `enum_tree` calls `enum_forest` only once and returns a continuation. This call to `enum_forest` is performed as soon as `enum_tree` receives two arguments, which is precisely the way it is used within `enum_forest`.

To run C2 on a forest `f`, one still applies `enum_forest` to `f` with a display continuation. The result is now itself a continuation, that must be invoked to perform the actual enumeration, as follows:

```
enum_forest (fun () → (* display current configuration *)) f ()
```

C2 makes more intensive use of higher-order functions than C1: we now employ functions that return functions. The principle remains the same, though: if the function `k` enumerates the colorings of the forest `f0`, then the function `enum_forest k f` (resp. `enum_tree k t`) enumerates those of the forest `f @ f0` (resp. `t :: f0`). One may notice that `enum_forest` and `enum_tree` are now instances of the generic “fold” functions associated to the data types `tree` and `forest`. Still, for the sake of clarity, we prefer to define them directly.

Complexity. The functions `enum_forest` and `enum_tree` now have three arguments. Applying them to one argument does not trigger any computation, but the second and third applications have distinct costs, which must be measured separately.

The cost of an application to two arguments is easily determined. Indeed, every node in the forest at hand is clearly traversed exactly once; furthermore, traversing every node induces a unit cost, due to the closure that is built within `enum_tree`. Hence, the total cost is the number of nodes, n. Moreover, because only this preliminary phase allocates memory, we may immediately conclude that C2's space complexity is $O(n)$.

The cost of a third application is measured as in the previous section. We now write $F(k, f)$ (resp. $T(k, t)$) for the cost of *executing* the function *obtained* by invoking `enum_forest` (resp. `enum_tree`) with a continuation of cost k. From the code C2, we derive the following equations:

$$F(k, \texttt{[]}) \quad = \quad k \tag{7}$$
$$F(k, t :: f) \quad = \quad T(F(k, f), t) \tag{8}$$
$$T(k, \texttt{Node}\ f) \quad = \quad 1 + k + F(k, f) \tag{9}$$

Only the second equation differs from those that describe C1. Given these equations, it is straightforward to verify the following identities:

$$F(k, f) \quad = \quad N(f) \times (k + 1) - 1$$
$$T(k, t) \quad = \quad N(t) \times (k + 1) - 1$$

Applying `enum_forest` to a forest `f` with a costless initial continuation has a cost of $n(f)$. Then, invoking the continuation thus obtained entails a cost of $N(f) - 1$. Since $n(f) \leqslant N(f)$ holds, we may conclude that C2 has time complexity $O(N)$, which is obviously optimal. The first phase above can be viewed as a "pre-planning" phase, which produces a network of continuations. Then, the second phase performs the actual enumeration, without allocating any new closures.

4 Second refinement: defunctionalizing

The algorithm given in the previous section has optimal cost. Yet, it is still possible to reap a small constant factor. Indeed, we notice that every continuation built by the code in Figure 4 contains calls to unknown functions, namely `k` and `lf`. The OCaml compiler represents these functions as *closures* containing a code pointer and a data environment. This may incur a speed penalty on modern processors, because jumps to unknown addresses often defeat the branch prediction unit, causing a pipeline stall. One way to address this problem is to replace the branch to an unknown address with a test, followed with a branch to a constant address. In other words, we will now abandon the use of higher-order functions. To replace them, we will introduce a data structure, together with a (first-order) function `run` which interprets its values as functions. This technique, known as *defunctionalization*, was introduced by Reynolds three decades ago (Reynolds, 1998a; Reynolds, 1998b). It has recently received some new interest as a program transformation (Danvy & Nielsen, 2001) or compilation (Cejtin *et al.*, 2000) technique. Indeed, the program transformation which we are about to describe could be performed automatically by a compiler such as MLton (Cejtin *et al.*, 2002).

It is easy to observe that every continuation manipulated by C2 is either the initial continuation (which displays the current configuration), or a continuation built by `enum_tree`, whose code then consists of the last six lines of Figure 4. The initial continuation only needs access to the global array `bits`, so we will assume that it has no free variables. Continuations of the latter kind, on the other hand, have three free variables, namely `i`, `k` and `lf`. This analysis leads us to the following data type definition:

```
type continuation =
  | Display
  | Continue of int × continuation × continuation
```

```
let rec enum_forest k = function
  | [] → k
  | t :: f → enum_tree (enum_forest k f) t
and enum_tree k (Node (i,f)) =
  Continue (i, k, enum_forest k f)

let rec run = function
  | Display →
      (* display current configuration *)
  | Continue (i, k, lf) →
      if bits.(i) = 0 then begin
        run k; bits.(i) ← 1; run lf
      end else begin
        run lf; bits.(i) ← 0; run k
      end
```

Fig. 5. Second refinement (C3).

A value of type `continuation` contains a tag – either `Display` or `Continue` – which effectively plays the role of a code pointer. When the tag is `Continue`, it is accompanied with values for `i`, `k` and `lf`, which suffice to capture the continuation's meaning.

The defunctionalized version of `enum_tree`, given in Figure 5, now returns a data structure of type `continuation`, instead of an actual continuation. To use such a data structure, we must interpret it as a function, that is, describe how it is "run". This is the role of the new function `run`. The function proceeds by cases, according to the continuation's tag. If it is `Display`, the current configuration is displayed (code omitted). If it is `Continue`, then suitable values for `i`, `k` and `lf` are read from the data structure, and the continuation's code is executed. It is taken from the last five lines of Figure 4, with calls to `k` and `lf` replaced with recursive calls to `run`. To run C3 on a forest `f`, one writes `run (enum_forest Display f)`.

According to measurements performed on a number of random forests, this refinement yields a performance increase that is consistently comprised between 20% and 30%. Although this may be deemed a rather small improvement, we found it interesting, in particular because this formulation helped us discover the next refinement.

5 Last refinement: using integer continuations

From the definition of `enum_tree` in Figure 5, it is now clear that `enum_forest k f` allocates exactly one `continuation` object for every node in the forest `f`. (One may also notice that these objects form a directed acylic graph.) So, the initial continuation set aside, continuations are in one-to-one correspondence with nodes. This prompts us to identify the two notions, and – considering nodes are numbered – to represent continuations as integers. By convention, the integer -1 will be used to represent the initial continuation.

What becomes of the information stored in `Continue` objects? The integer `i` becomes redundant, since it now *is* the continuation. The continuation `k` (resp. `lf`)

```
let rec enum_forest k = function
  | [] → k
  | t :: f → enum_tree (enum_forest k f) t
and enum_tree k (Node (i,f)) =
  ka.(i) ← k;
  lfa.(i) ← enum_forest k f;
  i

let rec run = function
  | (-1) →
      (* display current configuration *)
  | i →
      if bits.(i) = 0 then begin
        run ka.(i); bits.(i) ← 1; run lfa.(i)
      end else begin
        run lfa.(i); bits.(i) ← 0; run ka.(i)
      end
```

Fig. 6. Last refinement (C4).

will now be stored at index `i` in a global array `ka` (resp. `lfa`) of size n. Because continuations are now integers, `ka` and `lfa` are arrays of integers.

The new version of `enum_tree`, given in Figure 6, now initializes the arrays `ka` and `lfa` instead of allocating continuations, and returns `i` itself instead of a fresh `Continue` object. The algorithm's asymptotic space complexity remains unchanged, but a constant factor is saved, whose exact amount depends on the runtime system.

In `run`, the initial continuation is now distinguished by the special value -1. In the general case, `i` stands for a node number, and the two continuation nodes `k` and `lf` are obtained by looking up the arrays `ka` and `lfa` at index `i`. To run C4 on a forest `f`, one writes `run (enum_forest (-1) f)`.

According to measurements performed on a number of random forests, this refinement yields a performance increase that is consistently comprised between 0 and 10 percent. This is a minor improvement, but we believe this formulation is nevertheless interesting, for two reasons. First, it is amenable to a very simple implementation in a low-level language such as C. All storage is allocated in three global arrays, requiring no dynamic allocation. Secondly, it sheds some light on the algorithm's structure. Since a continuation is now either a node or -1, the arrays `ka` and `lfa` can be viewed as partial mappings from nodes to nodes. One may check that they are initialized by `enum_forest` and `enum_tree` as follows:

- If `i` is the root of the left-most tree in the forest, then `ka.(i)` is -1;
- if `i` has a left sibling `j` in the forest, then `ka.(i)` is `j`;
- otherwise, `i` must have a parent `j` in the forest, and `ka.(i)` is `ka.(j)`.
- If `i` has a child in the forest, then `lfa.(i)` is its right-most child;
- otherwise, `lfa.(i)` is `ka.(i)`.

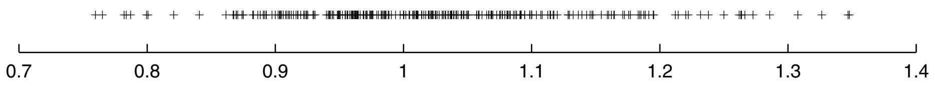
This version of the algorithm bears a rather strong resemblance with Knuth's coroutine-based algorithm (Knuth, 2001a). Indeed, Knuth's algorithm defines exactly one coroutine per node, and relies on tables which map every node to its left sibling and to its right-most child, if defined. However, Knuth's approach has an inherent deficiency: coroutines signal completion by returning, which may cause the whole call stack to be unwound, whereas they do so, in our case, by invoking a continuation. Thus, as recognized by Knuth, his algorithm may have asymptotically worse behavior in some cases. It is noteworthy that our approach naturally leads to an algorithm that is superficially similar to Knuth's, but easier to understand, and more efficient.

Knuth's "loopless" algorithm, which appears similar to Koda and Ruskey's original description (Koda & Ruskey, 1993), addresses this deficiency by using a mutable data structure that is significantly more complex. The next section compares it with ours.

6 Performance assessment

We now compare C4, performance-wise, with Knuth's "loopless" implementation L. Both were compiled to x86 machine code, using the native OCaml compiler with array bounds checking turned off, and `gcc -O2`, respectively. (We have also hand-translated C4 to C code, with no noticeable time difference with respect to the OCaml code.) L implements Koda and Ruskey's more efficient algorithm, which is loopless, that is, performs a constant amount of computation between two consecutive colorings. Our implementation is not loopless, but has the same overall time complexity, namely $O(N)$.

In practice, the two implementations seem to have very similar performance, as suggested by the following graph. Every data point shows the ratio of their running times (that is, C4's divided by L's) for a random forest (with $30 \leqslant n < 45$). The graph has three hundred data points. We have verified that this ratio does not appear to be correlated with n or N.



These measurements reflect the time necessary to *produce* the Gray code only—nothing was displayed. In a realistic application, every coloring would be *exploited* for some purpose before producing the next coloring, so the performance difference between the two implementations would be even less noticeable. In light of this remark, we believe it is safe to claim that the two implementations are equally efficient.

Our code is available electronically (Filliâtre & Pottier, 2002); it is functionally equivalent to Knuth's (Knuth, 2001a).

7 Conclusion

We have proposed a functional, higher-order implementation of Koda and Ruskey's algorithm. From it, we have derived a first-order version whose efficiency is comparable to Knuth's C implementation.

One key advantage of our continuation-based formulation (C2) is to be amenable to formal proof. It is possible to give reasonably simple specifications for `enum_tree` and `enum_forest`. Because these functions must enumerate colorings in either direction, this requires characterizing the final coloring of the Gray code sequence associated with a given forest. This can be done inductively over trees and forests. As a result, the formalization is rather straightforward to conduct within a proof assistant such as Coq (Barras *et al.*, 2002). We are currently in the process of carrying out such a task.

References

Barras, B., Herbelin, H. *et al.* (2002) *The Coq Proof Assistant*. URL: `http://coq.inria.fr/`.

Cejtin, H., Jagannathan, S. and Weeks, S. (2000) Flow-directed closure conversion for typed languages. In: Smolka, G., editor, *Proceedings 2000 European Symposium on Programming (ESOP'00): Lecture Notes in Computer Science 1782*, pp. 56–71. Springer Verlag.

Cejtin, H., Fluet, M., Jagannathan, S. and Weeks, S. (2002) *The MLton Standard ML Compiler*. URL: `http://www.mlton.org/`.

Danvy, O. and Nielsen, L. R. (2001) Defunctionalization at work. *Third International Conference on Principles and Practice of Declarative Programming (PPDP 2001)*. (Also available as BRICS Research Report RS-01-23. URL: `http://www.brics.dk/RS/01/23/BRICS-RS-01-23.ps.gz`.)

Filliâtre, J.-C. and Pottier, F. (2002) *Functional implementations of Koda and Ruskey's algorithm*. URL: `http://www.lri.fr/~filliatr/software.en.html`.

Knuth, D, E. (2001a) *An implementation of Koda and Ruskey's algorithm*. URL: `http://www-cs-staff.stanford.edu/~knuth/programs.html`.

Knuth, D. E. (2001b) *The Art of Computer Programming*. Addison-Wesley.

Koda, Y. and Ruskey, F. (1993) A Gray code for the ideals of a forest poset. *J. Algorithms*, **15**(2), 324–340.

Leroy, X., Doligez, D. *et al.* (2002) *The Objective Caml language*. URL: `http://caml.inria.fr/`.

Reynolds, J. C. (1998a) Definitional interpreters for higher-order programming languages. *Higher-order & Symbolic Computation*, **11**(4), 363–397.

Reynolds, J C. (1998b) Definitional interpreters revisited. *Higher-order & Symbolic Computation*, **11**(4), 355–361.