Higher-order imperative enumeration of binary trees in COQ

Sylvain Boulmé

June 1, 2007

This document describes some proofs about `enumBT`, a higher-order imperative function such that `enumBT n f` calls successively `f` over all and only binary trees of height `n`. Moreover, each tree is enumerated only once. I use the following COQ definition for binary trees:

```
Inductive bintree : Set :=
  | Leaf: bintree | Node: bintree -> bintree -> bintree.
```

In the following, I also use some definitions of the COQ library. Type `nat` is type of Peano numbers generated from `O` and `S`. Type `Z` is type of infinite binary integers (more efficient than `nat` to perform large concrete computations). At last, `list` is the polymorphic type of lists.

The COQ definition of `enumBT` is given below. Here, `K` is the "specification type" of the state DSM (see [Bou06]). It is parametrized both by `St` the type of the global state and by `unit` the type of the result. It uses an infix operator `-;` to denote sequences: "`p1 -; p2`" is a notation for "`bind p1 (fun _:unit => p2)`". The main advantage of this CPS-like implementation is to call `f` as soon as a tree is computed, before to compute the next tree. Moreover, whereas the number of binary trees is exponential in function of 2^n (e.g. the number of nodes of a balanced binary tree of height n), this function requires only a memory linear in function of 2^n.

```
Fixpoint enumBT (St:Type) (n:nat) (f:bintree -> K St unit) {struct n}
  : K St unit := match n with
  | O => f Leaf
  | (S p) =>
      enumBT p (fun l => enumBT p (fun r => f (Node l r)) -;
                         enumlt p (fun r => f (Node l r) -;
                                            f (Node r l)))
  end
with enumlt (St:Type) (n:nat) (f:bintree -> K St unit) {struct n}
  : K St unit := match n with
  | O => skip
  | (S p) => (enumBT p f) -; (enumlt p f)
  end.
```

Function `enumBT` is defined mutually recursively over `n` with `enumlt` which enumerates binary trees with a height strictly lower than `n`. Then, it uses the fact that in a tree of height (S n), either its two children have a height equal to n, or one of them has a height equals to n and the other has a height strictly lower than n.

Induction method for enumBT and enumlt The following lines explain how my proof deals with the mutually recursive definition of enumBT and enumlt. Assuming St:**Type**, and given two predicate P and Q of type

```
nat -> ((bintree -> K St unit) -> (K St unit)) -> Prop
```

such that I want to prove enumBT_P and enumlt_Q formulae given below. I first prove enumlt_Q_aux below by structural induction over n. Then, I prove enumBT_P using induction lemma nat_le_ind given below. At last, enum_lt_Q is trivially derived from the two previous lemma.

```
enumlt_Q_aux: forall (n:nat),
    (forall (m:nat), m < n -> P m (enumBT m)) -> Q n (enumlt n).

enumBT_P: forall (n:nat), P n (enumBT n).

enumlt_Q: forall (n:nat), Q n (enumlt n).
```

Property nat_le_ind is a variant of well-founded induction over natural numbers.

```
Lemma nat_le_ind: forall (P:nat -> Prop),
  (P O)
   -> (forall (n:nat), (forall (m:nat), m <= n -> P m) -> P (S n))
     -> forall (n:nat), P n.
```

Typically, the monotonicity of enumBT is proved by this way.

```
Lemma enumBT_monotonic:
  forall (St: Type) (n:nat) (p1 p2: bintree -> K St unit) ,
    (forall (t:bintree) (st:St), refInEnv st (p1 t) (p2 t))
     -> forall (st:St), (refInEnv st (enumBT n p1) (enumBT n p2)).
```

Number of binary trees generated by enumBT In this paragraph, I prove that for a given n, enumBT generates (numBT n) binary trees where numBT is defined from num2 below. The first component computed by (num2 h) is the number of binary trees of height h, and the second component computed by (num2 h) is the number of binary trees with a height lower than h.

```
Fixpoint num2 (h: nat) {struct h}: Z*Z :=
  match h with
  | O => (1,0)
  | (S h1) =>
         let (nh1,nlh1) := (num2 h1) in
         let nlh := nh1 + nlh1 in
            (nh1*(nlh+nlh1),nlh)
  end.

Definition numBT (h:nat) : Z := fst (num2 h).
```

Of course, this function explodes. For (numBT 4), Coq computes 651. For (numBT 8), it computes 1947270476915296449559659317606103024276803403.

Now, I define incr such that incr n adds n to a global integer variable :

```
Definition incr (n:Z) : K Z unit := bind (get Z) (fun x:Z => set (x+n)).
```

The announced result is expressed by the following theorem:

```
Theorem enum_incr: forall (h:nat)   (init:Z),
  refInEnv init (enumBT h (fun t => incr 1))  (incr (numBT h)).
```

Before to prove this theorem, we can automatically check it for some concrete values of h and init. Hence, the goal with h being 5 and init being 0 (e.g. refInEnv 0 (enumBT 5 (fun t => incr 1)) (incr (numBT 5))) is reduced by wp-computation in less than one second into True->(457653, tt)=(457653, tt). This illustrate the efficiency of the new Coq virtual machine (see [Gré02]), because computing (enumBT 5 (fun t => incr 1)) reduces to compute a sequence of 457653 "inc 1".

Actually, the proof of theorem enum_incr is a trivial application of enum_incr_gen below.

```
Lemma enum_incr_gen: forall (h:nat) (acc init:Z),
  refInEnv init (enumBT h (fun t => incr acc))  (incr ((numBT h)*acc)).
```

The proof of enum_incr_gen uses the induction method described above. It is easy (generated by a script of about 30 Coq commands) and mainly combines transitivity and monotonicity of refinement, with the ring structure of Z and the following property of incr:

```
Lemma incr_seq: forall (n m init:Z),
  refInEnv init  ((incr n) -; (incr m))  (incr (n+m)).
```

General specification of enumBT The general specification uses a predicate enumlist: forall (A:Set), (A -> Prop) -> (list A) -> Prop such that enumlist P l expresses that l contains only all elements of A satisfying P without duplicates. It also uses height: bintree -> nat computing the height of a binary tree. At last, it uses genBT: nat -> (list bintree) such that

```
Theorem genBT_enumlist: forall (n:nat),
  (enumlist (fun t => (height t)=n) (genBT n)).
```

Then, defining

```
Fixpoint revIter (St:Type) (A:Set) (l:list A) (p:A->K St unit) {struct l}
  : (K St unit) := match l with
     | nil => skip
     | x::m => revIter m p -; p x
     end.
```

and, using the previous induction method, I have easily proved that:

```
Theorem enumBT_spec:
  forall (St:Type) (h:nat) (f: bintree -> K St unit) (st:St),
    refInEnv st (enumBT h f) (listRevIter (genBT h) f).
```

Hence, the difficult part here is to establish theorem `genBT_enumlist`. At last, let me remark that `enum_incr` could be probably derived from `enumBT_spec`, but it supposes to prove the corresponding property about `genBT` which seems not easier than proving `enum_incr` directly.

Conclusion All the results proved here could be actually proved for a state monad using observational equivalence instead of refinement. Indeed, this whole example uses only the state monad fragment of the state DSM. Here, I used refinement relation through `refInEnv`, because in the current state of my implementation, I have not defined an `eqInEnv` relation, nor have I defined simplification rules to reason about observational equivalence.

References

[Bou06] S. Boulmé. *Higher-Order Refinement In Coq (reports and Coq files).* Web page: `http://www-lsr.imag.fr/users/Sylvain.Boulme/horefinement/`, 2006.

[Gré02] B. Grégoire and X. Leroy. A compiled implementation of strong reduction. In *Proc. of the ACM SIGPLAN ICFP'02*, 235–246. ACM Press, New York, NY, USA, 2002.