EXTRACT

RUNNING COQ PROGRAMS IN ML

```
Require Extraction.
```

Coq's `Extraction` feature allows you to write a functional program inside Coq; (presumably) use Coq's logic to prove some correctness properties about it; then print it out as an ML (or Haskell) program that you can compile with your optimizing ML (or Haskell) compiler.

The Extraction chapter of *Logical Foundations* gave a simple example of Coq's program extraction features. In this chapter, we'll take a deeper look.

```
Set Warnings "-extraction-inside-module". (* turn off a warning message *)
Require Import Perm.

Module Sort1.
Fixpoint insert (i:nat) (l: list nat) :=
  match l with
  | nil ⇒ i::nil
  | h::t ⇒ if i <=? h then i::h::t else h :: insert i t
 end.

Fixpoint sort (l: list nat) : list nat :=
  match l with
  | nil ⇒ nil
  | h::t ⇒ insert h (sort t)
end.
```

The `Extraction` command prints out a function as Ocaml code.

```
Require Coq.extraction.Extraction.
Extraction sort.
```

You can see the translation of "sort" from Coq to Ocaml, in the "Messages" window of your IDE. Examine it there, and notice the similarities and differences.

However, we really want the whole program, including the `insert` function. We get that as follows:

```
Recursive Extraction sort.
```

The first thing you see there is a redefinition of the `bool` type. But Ocaml already has a `bool` type whose inductive structure is isomorphic. We want our extracted functions to be compatible with, callable by, ordinary Ocaml code. So we want to use Ocaml's standard notation for the inductive definition, `bool`. The following directive accomplishes that:

```
Extract Inductive bool ⇒ "bool" [ "true" "false" ].
Extract Inductive list ⇒ "list" [ "[]" "(::)" ].
Recursive Extraction sort.

End Sort1.
```

This is better. But the program still uses a unary representation of natural numbers: the number 7 is really (S (S (S (S (S (S (S O)))))). which in Ocaml will be a data structure that's seven pointers deep. The `leb` function takes time proportional to the difference in value between `n` and `m`, which is terrible. We'd like natural numbers to be represented as Ocaml `int`. Unfortunately, there are only a finite number of `int` values in Ocaml (2^31, or 2^63, depending on your implementation); so there are things you could prove about some programs, in Coq, that wouldn't be true in Ocaml.

There are two solutions to this problem:

- Instead of using `nat`, use a more efficient constructive type, such as `Z`.
- Instead of using `nat`, use an *abstract type*, and instantiate it with Ocaml integers.

The first alternative uses Coq's `Z` type, an inductive type with constructors `xI` `xH` etc. `Z` represents 7 as `Zpos (xI (xI xH))`, that is, +(1+2*(1+2* 1)). A number `n` is represented as a data structure of size log(n), and the operations (plus, less-than) also take about log(n) each.

`Z`'s log-time per operation is much better than linear time; but in Ocaml we are used to having constant-time operations. Thus, here we will explore the second alternative: program with abstract types, then use an extraction directive to get efficiency.

```
Require Import ZArith.
Open Scope Z_scope.
```

We will be using `Parameter` and `Axiom` in Coq. You already saw these keywords, in a `Module Type`, in the ADT chapter. There, they describe interface components that must be instantiated by any `Module` that satisfies the type. Here, we will use this feature in a different (and more dangerous) way: To axiomatize a mathematical theory without actually constructing it. The reason that's dangerous is that if your axioms are inconsistent, then you can prove `False`, or in fact, you can prove *anything*, so all your proofs are worthless. So we must take care!

Here, we will axiomatize a *very weak* mathematical theory: We claim that there exists some type `int` with a function `ltb`, so that `int` injects into `Z`, and `ltb` corresponds to the < relation on Z. That seems true enough (for example, take `int=Z`), but we're not *proving* it here.

```
Parameter int : Type. (* This is the Ocaml int type. *)
Extract Inlined Constant int ⇒ "int". (* so, extract it that way! *)

Parameter ltb: int → int → bool. (* This is the Ocaml (<) operator. *)
Extract Inlined Constant ltb ⇒ "(<)". (* so, extract it that way! *)
```

Now, we need to axiomatize `ltb` so that we can reason about programs that use it. We need to take great care here: the axioms had better be consistent with Ocaml's behavior, otherwise our proofs will be meaningless.

One axiomatization of `ltb` is just that it's a total order, irreflexive and transitive. This would work just fine. But instead, I choose to claim that there's an injection from "int" into the mathematical integers, Coq's `Z` type. The reason to do this is then we get to use the `omega` tactic, and other Coq libraries about integer comparisons.

```
Parameter int2Z: int → Z.
Axiom ltb_lt : ∀ n m : int, ltb n m = true ↔ int2Z n < int2Z m.
```

Both of these axioms are sound. There does (abstractly) exist a function from "int" to `Z`, and that function *is* consistent with the `ltb_lt` axiom. But you should think about this until you are convinced.

Notice that we do not give extraction directives for `int2Z` or `ltb_lt`. That's because they will not appear in *programs*, only in proofs that are not meant to be extracted.

Now, here's a dangerous axiom:

```
Parameter ocaml_plus : int → int → int.

Extract Inlined Constant ocaml_plus ⇒ "(+)".

Axiom ocaml_plus_plus: ∀a b c: int, ocaml_plus a b = c ↔ int2Z a + int2Z b = int2Z c.
```

The first two lines are OK: there really is a "+" function in Ocaml, and its type really is `int → int → int`.

But `ocaml_plus_plus` is unsound! From it, you could prove,

`(int2Z max_int + int2Z max_int) = int2Z (ocaml_plus max_int max_int)`,

which is not true in Ocaml, because overflow wraps around, modulo `2^(wordsize-1)`.

So we won't axiomatize Ocaml addition.

Utilities for OCaml Integer Comparisons

Just like in Perm.v, but for `int` and `Z` instead of `nat`.

```
Lemma int_blt_reflect : ∀ x y, reflect (int2Z x < int2Z y) (ltb x y).
Proof.
  intros x y.
  apply iff_reflect. symmetry. apply ltb_lt.
Qed.

Lemma Z_eqb_reflect : ∀ x y, reflect (x=y) (Z.eqb x y).
Proof.
  intros x y.
  apply iff_reflect. symmetry. apply Z.eqb_eq.
Qed.

Lemma Z_ltb_reflect : ∀ x y, reflect (x<y) (Z.ltb x y).
Proof.
  intros x y.
  apply iff_reflect. symmetry. apply Z.ltb_lt.
Qed.

(* Add these three lemmas to the Hint database for bdestruct,
  so the bdestruct tactic will work with them. *)
Hint Resolve int_blt_reflect Z_eqb_reflect Z_ltb_reflect : bdestruct.
```

SearchTrees, Extracted

Let us re-do binary search trees, but with Ocaml integers instead of Coq nats.

Maps, on `Z` Instead of `nat`

Our original proof with nats used `Maps.total_map` in its abstraction relation, but that won't work here because we need maps over `Z` rather than `nat`. So, we copy-paste-edit to make `total_map` over `Z`.

```
Require Import Coq.Logic.FunctionalExtensionality.

Module IntMaps.
Definition total_map (A:Type) := Z → A.
Definition t_empty {A:Type} (v : A) : total_map A := (fun _ ⇒ v).
Definition t_update {A:Type} (m : total_map A) (x : Z) (v : A) :=
  fun x' ⇒ if Z.eqb x x' then v else m x'.
Lemma t_update_eq : ∀ A (m: total_map A) x v, (t_update m x v) x = v.
 +

Theorem t_update_neq : ∀ (X:Type) v x1 x2 (m : total_map X),
  x1 ≠ x2 → (t_update m x1 v) x2 = m x2.
 +
```

```
Lemma t_update_shadow : ∀ A (m: total_map A) v1 v2 x,
    t_update (t_update m x v1) x v2 = t_update m x v2.
+

End IntMaps.

Import IntMaps.
```

Trees, on `int` Instead of `nat`

```
Module SearchTree2.
Section TREES.
Variable V : Type.
Variable default: V.

Definition key := int.

Inductive tree : Type :=
 | E : tree
 | T: tree → key → V → tree → tree.

Definition empty_tree : tree := E.

Fixpoint lookup (x: key) (t : tree) : V :=
  match t with
   | E ⇒ default
   | T tl k v tr ⇒ if ltb x k then lookup x tl
                         else if ltb k x then lookup x tr
                         else v
  end.

Fixpoint insert (x: key) (v: V) (s: tree) : tree :=
 match s with
 | E ⇒ T E x v E
 | T a y v' b ⇒ if ltb x y then T (insert x v a) y v' b
                       else if ltb y x then T a y v' (insert x v b)
                       else T a x v b
 end.

Fixpoint elements' (s: tree) (base: list (key*V)) : list (key * V) :=
 match s with
 | E ⇒ base
 | T a k v b ⇒ elements' a ((k,v) :: elements' b base)
 end.

Definition elements (s: tree) : list (key * V) := elements' s nil.

Definition combine {A} (pivot: Z) (m1 m2: total_map A) : total_map A :=
  fun x ⇒ if Z.ltb x pivot then m1 x else m2 x.

Inductive Abs: tree → total_map V → Prop :=
| Abs_E: Abs E (t_empty default)
| Abs_T: ∀ a b l k v r,
      Abs l a →
      Abs r b →
      Abs (T l k v r) (t_update (combine (int2Z k) a b) (int2Z k) v).

Theorem empty_tree_relate: Abs empty_tree (t_empty default).
Proof.
constructor.
Qed.
```

Exercise: 3 stars (lookup_relate)

```
Theorem lookup_relate:
  ∀ k t cts , Abs t cts → lookup k t = cts (int2Z k).
```

```
  Proof. (* Copy your proof from SearchTree.v, and adapt it. *)
  (* FILL IN HERE *) Admitted.
```
□

Exercise: 3 stars (insert_relate)

```
  Theorem insert_relate:
   ∀ k v t cts,
      Abs t cts →
      Abs (insert k v t) (t_update cts (int2Z k) v).
  Proof. (* Copy your proof from SearchTree.v, and adapt it. *)
  (* FILL IN HERE *) Admitted.
```
□

Exercise: 1 star (unrealistically_strong_can_relate)

```
  Lemma unrealistically_strong_can_relate:
   ∀ t, ∃ cts, Abs t cts.
  Proof. (* Copy-paste your proof from SearchTree.v; it should work as is. *)
  (* FILL IN HERE *) Admitted.
```
□

```
  End TREES.
```

Now, run this command and examine the results in the "results" window of your IDE:

```
  Recursive Extraction empty_tree insert lookup elements.
```

Next, we will extract it into an Ocaml source file, and measure its performance.

```
  Extraction "searchtree.ml" empty_tree insert lookup elements.
```

Note: we've done the extraction *inside* the module, even though Coq warns against it, for a specific reason: We want to extract only the program, not the proofs.

```
  End SearchTree2.
```

Performance Tests

Read the Ocaml program, test_searchtree.ml:

```
let test (f: int -> int) (n: int) =
 let rec build (j, t) = if j=0 then t
                    else build(j-1, insert (f j) 1 t)
  in let t1 = build(n,empty_tree)
  in let rec g (j,count) = if j=0 then count
                       else if lookup 0 (f j) t1 = 1
                            then g(j-1,count+1)
                            else g(j-1,count)
  in let start = Sys.time()
  in let answer = g(n,0)
      in (answer, Sys.time() -. start)

let print_test name (f: int -> int) n =
  let (answer, time) = test f n
  in (print_string "Insert and lookup "; print_int n;
      print_string " "; print_string name; print_string " integers in ";
      print_float time; print_endline " seconds.")

let test_random n = print_test "random" (fun _ -> Random.int n) n
```

```
let test_consec n = print_test "consecutive" (fun i -> n-i) n

let run_tests() = (test_random 1000000; test_random 20000; test_consec 20000)

let _ = run_tests ()
```

You can run this inside the ocaml top level by:

```
#use "searchtree.ml";;
#use "test_searchtree.ml";;
run_tests();;
```

On my machine, in the byte-code interpreter this prints,

```
Insert and lookup 1000000 random integers in 1.076 seconds.
Insert and lookup 20000 random integers in 0.015 seconds.
Insert and lookup 20000 consecutive integers in 5.054 seconds.
```

You can compile and run this with the ocaml native-code compiler by:

```
ocamlopt searchtree.mli searchtree.ml -open Searchtree test_searchtree.ml -o test_searchtree
./test_searchtree
```

On my machine this prints,

```
Insert and lookup 1000000 random integers in 0.468 seconds.
Insert and lookup 20000 random integers in 0. seconds.
Insert and lookup 20000 consecutive integers in 0.374 seconds.
```

Unbalanced Binary Search Trees

Why is the performance of the algorithm so much worse when the keys are all inserted consecutively? To examine this, let's compute with some searchtrees inside Coq. We cannot do this with the search trees defined thus far in this file, because they use a key-comparison function `ltb` that is abstract and uninstantiated (only during Extraction to Ocaml does `ltb` get instantiated).

So instead, we'll use the SearchTree module, where everything runs inside Coq.

```
Require SearchTree.
Module Experiments.
Open Scope nat_scope.
Definition empty_tree := SearchTree.empty_tree nat.
Definition insert := SearchTree.insert nat.
Definition lookup := SearchTree.lookup nat 0.
Definition E := SearchTree.E nat.
Definition T := SearchTree.T nat.

Goal insert 5 1 (insert 4 1 (insert 3 1 (insert 2 1 (insert 1 1 (insert 0 1
empty_tree))))) = E.
simpl. fold E; repeat fold T.
```

Look here! The tree is completely unbalanced. Looking up 5 will take linear time. That's why the runtime on consecutive integers is so bad.

```
Abort.
```

Balanced Binary Search Trees

To achieve robust performance (that stays N log N for a sequence of N operations, and does not degenerate to N*N), we must keep the search trees balanced. The next chapter, Redblack, implements that idea.

```
End Experiments.
```