TRIE

NUMBER REPRESENTATIONS AND EFFICIENT LOOKUP TABLES

LogN Penalties in Functional Programming

Purely functional algorithms sometimes suffer from an asymptotic slowdown of order logN compared to imperative algorithms. The reason is that imperative programs can do *indexed array update* in constant time, while functional programs cannot.

Let's take an example. Give an algorithm for detecting duplicate values in a sequence of N integers, each in the range 0..2N. As an imperative program, there's a very simple linear-time algorithm:

```
collisions=0;
for (i=0; i<2N; i++)
   a[i]=0;
for (j=0; j<N; j++) {
   i = input[j];
   if (a[i] != 0)
     collisions++;
   a[i]=1;
}
return collisions;
```

In a functional program, we must replace `a[i]=1` with the update of a finite map. If we use the inefficient maps in `Maps.v`, each lookup and update will take (worst-case) linear time, and the whole algorithm is quadratic time. If we use balanced binary search trees `Redblack.v`, each lookup and update will take (worst-case) logN time, and the whole algorithm takes NlogN. Comparing O(NlogN) to O(N), we see that there is a logN asymptotic penalty for using a functional implementation of finite maps. This penalty arises not only in this "duplicates" algorithm, but in any algorithm that relies on random access in arrays.

One way to avoid this problem is to use the imperative (array) features of a not-really-functional language such as ML. But that's not really a functional program! In particular, in *Verified Functional Algorithms* we prove program correct by relying on the *tractable proof theory* of purely functional programs; if we use nonfunctional features of ML, then this style of proof will not work. We'd have to use something like Hoare logic instead (see `Hoare.v` in volume 2 of *Software Foundations*), and that is not *nearly* as nice.

Another choice is to use a purely functional programming language designed for imperative programming: Haskell with the IO monad. The IO monad provides a pure-functional interface to efficient random-access arrays. This might be a reasonable approach, but we will not cover it here.

Here, we accept the logN penalty, and focus on making the "constant factors" small: that is, let us at least have efficient functional finite maps.

Extract showed one approach: use Ocaml integers. The advantage: constant-time greater-than comparison. The disadvantages: (1) Need to make sure you axiomatize them correctly in Coq, otherwise your proofs are unsound. (2) Can't easily axiomatize addition, multiplication, subtraction, because Ocaml integers don't behave like the "mathematical" integers upon 31-bit (or 63-bit) overflow. (3) Can *only* run the programs in Ocaml, not inside Coq.

So let's examine another approach, which is quite standard inside Coq: use a construction in Coq of arbitrary-precision binary numbers, with logN-time addition, subtraction, and comparison.

A Simple Program That's Waaaaay Too Slow.

```
Require Import Perm.
Require Import Maps.
Import FunctionalExtensionality.

Module VerySlow.

Fixpoint loop (input: list nat) (c: nat) (table: total_map bool)
: nat :=
  match input with
  | nil ⇒ c
  | a::al ⇒ if table a
                 then loop al (c+1) table
                 else loop al c (t_update table a true)
 end.

Definition collisions (input: list nat) : nat :=
       loop input 0 (t_empty false).
```

```
Example collisions_pi: collisions [3;1;4;1;5;9;2;6] = 1.
Proof. reflexivity. Qed.
```

This program takes cubic time, O(N^3). Let's assume that there are few duplicates, or none at all. There are `N` iterations of `loop`, each iteration does a `table` lookup, most iterations do a `t_update` as well, and those operations each do `N` comparisons. The average length of the `table` (the number of elements) averages only N/2, and (if there are few duplicates) the lookup will have to traverse the entire list, so really in each iteration there will be only `N/2` comparisons instead of `N`, but in asymptotic analysis we ignore the constant factors.

So far it seems like this is a quadratic-time algorithm, O(N^2). But to compare Coq natural numbers for equality takes O(N) time as well:

```
Print beq_nat.
 (* fix beq_nat (n m : nat) {struct n} : bool :=
  match n with
  | 0 => match m with 0 => true | S _ => false end
  | S n1 => match m with 0 => false | S m1 => beq_nat n1 m1 end
  end *)
```

Remember, `nat` is a unary representation, with a number of `S` constructors proportional to the number being represented!

```
End VerySlow.
```

Efficient Positive Numbers

We can do better; we *must* do better. In fact, Coq's integer type, called `Z`, is a binary representation (not unary), so that operations such as `plus` and `leq` take time linear in the number of bits, that is, logarithmic in the value of the numbers. Here we will explore how `Z` is built.

```
Module Integers.
```

We start with positive numbers.

```
Inductive positive : Set :=
  | xI : positive → positive
  | xO : positive → positive
  | xH : positive.
```

A positive number is either

- 1, that is, `xH`
- 0+2n, that is, `xO n`
- 1+2n, that is, `xI n`.

For example, ten is 0+2(1+2(0+2(1))).

```
Definition ten := xO (xI (xO xH)).
```

To interpret a `positive` number as a `nat`,

```
Fixpoint positive2nat (p: positive) : nat :=
  match p with
  | xI q ⇒ 1 + 2 * positive2nat q
  | xO q ⇒ 0 + 2 * positive2nat q
  | xH ⇒ 1
 end.

Eval compute in positive2nat ten. (* = 10 : nat *)
```

We can read the binary representation of a positive number as the *backwards* sequence of `xO` (meaning 0) and `xI`/`xH` (1). Thus, ten is 1010 in binary.

```
Fixpoint print_in_binary (p: positive) : list nat :=
  match p with
  | xI q ⇒ print_in_binary q ++ [1]
  | xO q ⇒print_in_binary q ++ [0]
  | xH ⇒ [1]
 end.

Eval compute in print_in_binary ten. (*  = 1; 0; 1; 0 *)
```

Another way to see the "binary representation" is to make up postfix notation for `xI` and `xO`, as follows

```
Notation "p ¬ 1" := (xI p)
 (at level 7, left associativity, format "p '¬' '1'").
Notation "p ¬ 0" := (xO p)
 (at level 7, left associativity, format "p '¬' '0'").

Print ten. (* = xH~0~1~0 : positive *)
```

Why are we using positive numbers anyway? Since the zero was invented 2300 years ago by the Babylonians, it's sort of old-fashioned to use number systems that start at 1.

The answer is that it's highly inconvenient to have number systems with several different representations of the same number. For one thing, we don't want to worry about 00110=110. Then, when we extend this to the integers, with a "minus sign", we don't have to worry about -0 = +0.

To find the successor of a binary number—that is to increment— we work from low-order to high-order, until we hit a zero bit.

```
Fixpoint succ x :=
  match x with
    | p¬1 ⇒ (succ p)¬0
    | p¬0 ⇒ p¬1
    | xH ⇒ xH¬0
  end.
```

To add binary numbers, we work from low-order to high-order, keeping track of the carry.

```
Fixpoint addc (carry: bool) (x y: positive) {struct x} :
positive :=
  match carry, x, y with
    | false, p¬1, q¬1 ⇒ (addc true p q)¬0
    | false, p¬1, q¬0 ⇒ (addc false p q)¬1
    | false, p¬1, xH ⇒ (succ p)¬0
    | false, p¬0, q¬1 ⇒ (addc false p q)¬1
    | false, p¬0, q¬0 ⇒ (addc false p q)¬0
    | false, p¬0, xH ⇒ p¬1
    | false, xH, q¬1 ⇒ (succ q)¬0
    | false, xH, q¬0 ⇒ q¬1
    | false, xH, xH ⇒ xH¬0
    | true, p¬1, q¬1 ⇒ (addc true p q)¬1
    | true, p¬1, q¬0 ⇒ (addc true p q)¬0
    | true, p¬1, xH ⇒ (succ p)¬1
    | true, p¬0, q¬1 ⇒ (addc true p q)¬0
    | true, p¬0, q¬0 ⇒ (addc false p q)¬1
    | true, p¬0, xH ⇒ (succ p)¬0
    | true, xH, q¬1 ⇒ (succ q)¬1
    | true, xH, q¬0 ⇒ (succ q)¬0
    | true, xH, xH ⇒ xH¬1
  end.

Definition add (x y: positive) : positive := addc false x y.
```

Exercise: 2 stars (succ_correct)

```
Lemma succ_correct: ∀ p,
   positive2nat (succ p) = S (positive2nat p).
Proof.
(* FILL IN HERE *) Admitted.
```

☐

Exercise: 3 stars (addc_correct)

You may use `omega` in this proof if you want, along with induction of course. But really, using `omega` is an anachronism in a sense: Coq's `omega` uses theorems about `Z` that are proved from theorems about Coq's standard-library `positive` that, in turn, rely on a theorem much like this one. So the authors of the Coq standard library had to do the associative-commutative rearrangement proofs "by hand." But really, here you can use `omega` without penalty.

```
Lemma addc_correct: ∀ (c: bool) (p q: positive),
   positive2nat (addc c p q) =
        (if c then 1 else 0) + positive2nat p + positive2nat q.
Proof.
(* FILL IN HERE *) Admitted.

Theorem add_correct: ∀ (p q: positive),
   positive2nat (add p q) = positive2nat p + positive2nat q.
Proof.
intros.
unfold add.
apply addc_correct.
Qed.
```

☐

Claim: the `add` function on positive numbers takes worst-case time proportional to the log base 2 of the result.

We can't prove this in Coq, since Coq has no cost model for execution. But we can prove it informally. Notice that `addc` is structurally recursive on `p`, that is, the number of recursive calls is at most the height of the `p` structure; that's equal to log base 2 of `p` (rounded up to the nearest integer). The last call may call `succ q`, which is structurally recursive on `q`, but this `q` argument is what remained of the original `q` after stripping off a number of constructors equal to the height of `p`.

To implement comparison algorithms on positives, the recursion (Fixpoint) is easier to implement if we compute not only "less-than / not-less-than", but actually, "less / equal / greater". To express these choices, we use an Inductive data type.

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

Exercise: 5 stars (compare_correct)

```
Fixpoint compare x y {struct x}:=
  match x, y with
    | p¬1, q¬1 ⇒ compare p q
    | p¬1, q¬0 ⇒ match compare p q with Lt ⇒ Lt | _ ⇒ Gt end
    | p¬1, xH ⇒ Gt

  (* DELETE THIS CASE!  Replace it with cases that actually work. *)
    | _, _ ⇒ Lt
  end.

Lemma positive2nat_pos:
 ∀ p, positive2nat p > 0.
Proof.
intros.
induction p; simpl; omega.
Qed.

Theorem compare_correct:
 ∀ x y,
  match compare x y with
  | Lt ⇒ positive2nat x < positive2nat y
  | Eq ⇒ positive2nat x = positive2nat y
  | Gt ⇒ positive2nat x > positive2nat y
 end.
Proof.
induction x; destruct y; simpl.
(* FILL IN HERE *) Admitted.
```

☐

Claim: `compare x y` takes time proportional to the log base 2 of `x`. Proof: it's structurally inductive on the height of `x`.

Coq's Integer Type, `Z`

Coq's integer type is constructed from positive numbers:

```
Inductive Z : Set :=
  | Z0 : Z
  | Zpos : positive → Z
  | Zneg : positive → Z.
```

We can construct efficient (logN time) algorithms for operations on `Z`: `add`, `subtract`, `compare`, and so on. These algorithms call upon the efficient algorithms for `positives`.

We won't show these here, because in this chapter we now turn to efficient maps over positive numbers.

```
End Integers. (* Hide away our experiments with positive *)
```

These types, `positive` and `Z`, are part of the Coq standard library. We can access them here, because (above) the `Import Perm` has also exported `ZArith` to us.

```
Print positive. (* from the Coq standard library:
  Inductive positive : Set :=
  |  xI : positive -> positive
  | xO : positive -> positive
  | xH : positive *)

Check Pos.compare. (*   : positive -> positive -> comparison *)
Check Pos.add. (* : positive -> positive -> positive *)

Check Z.add. (* : Z -> Z -> Z *)
```

From `N*N*N` to `N*N*logN`

This program runs in `(N^2)*(log N)` time. The `loop` does `N` iterations; the table lookup does `O(N)` comparisons, and each comparison takes `O(log N)` time.

```
Module RatherSlow.

Definition total_mapz (A: Type) := Z → A.

Definition empty {A:Type} (default: A) : total_mapz A := fun _ ⇒
default.
Definition update {A:Type} (m : total_mapz A)
                    (x : Z) (v : A) :=
  fun x' ⇒ if Z.eqb x x' then v else m x'.

Fixpoint loop (input: list Z) (c: Z) (table: total_mapz bool) :
Z :=
  match input with
  | nil ⇒ c
  | a::al ⇒ if table a
                 then loop al (c+1) table
                 else loop al c (update table a true)
 end.

Definition collisions (input: list Z) := loop input 0 (empty
false).

Example collisions_pi: collisions [3;1;4;1;5;9;2;6]%Z = 1%Z.
Proof. reflexivity. Qed.
```

```
End RatherSlow.
```

From `N*N*logN` to `N*logN*logN`

We can use balanced binary search trees (red-black trees), with keys of type `Z`. Then the `loop` does `N` iterations; the table lookup does `O(logN)` comparisons, and each comparison takes `O(log N)` time. Overall, the asymptotic run time is `N*(logN)^2`.

Tries: Efficient Lookup Tables on Positive Binary Numbers

Binary search trees are very nice, because they can implement lookup tables from *any* totally ordered type to any other type. But when the type of keys is known specifically to be (small-to-medium size) integers, then we can use a more specialized representation.

By analogy, in imperative programming languages (C, Java, ML), when the index of a table is the integers in a certain range, you can use arrays. When the keys are not integers, you have to use something like hash tables or binary search trees.

A *trie* is a tree in which the edges are labeled with letters from an alphabet, and you look up a word by following edges labeled by successive letters of the word. In fact, a trie is a special case of a Deterministic Finite Automaton (DFA) that happens to be a tree rather than a more general graph.

A *binary trie* is a trie in which the alphabet is just {0,1}. The "word" is a sequence of bits, that is, a binary number. To look up the "word" 10001, use 0 as a signal to "go left", and 1 as a signal to "go right."

The binary numbers we use will be type `positive`:

```
Print positive.
(* Inductive positive : Set :=
    xI : positive -> positive
  | xO : positive -> positive
  | xH : positive *)

Goal 10%positive = xO (xI (xO xH)).
Proof. reflexivity. Qed.
```

Given a `positive` number such as ten, we will go left to right in the `xO/xI/` `constructors (which is from the low-order bit to the high-order bit), using [xO] as a signal to go left, [xI] as a signal to go right, and [xH] as a signal to stop.`

```
Inductive trie (A : Type) :=
    | Leaf : trie A
    | Node : trie A → A → trie A → trie A.
Arguments Leaf {A}.
Arguments Node {A} _ _ _.
```

```
Definition trie_table (A: Type) : Type := (A * trie A)%type.

Definition empty {A: Type} (default: A) : trie_table A :=
      (default, Leaf).

Fixpoint look {A: Type} (default: A) (i: positive) (m: trie A):
A :=
    match m with
    | Leaf ⇒ default
    | Node l x r ⇒
        match i with
        | xH ⇒ x
        | xO i' ⇒ look default i' l
        | xI i' ⇒ look default i' r
        end
    end.

Definition lookup {A: Type} (i: positive) (t: trie_table A) : A
:=
   look (fst t) i (snd t).

Fixpoint ins {A: Type} default (i: positive) (a: A) (m: trie A):
trie A :=
    match m with
    | Leaf ⇒
        match i with
        | xH ⇒ Node Leaf a Leaf
        | xO i' ⇒ Node (ins default i' a Leaf) default Leaf
        | xI i' ⇒ Node Leaf default (ins default i' a Leaf)
        end
    | Node l o r ⇒
        match i with
        | xH ⇒ Node l a r
        | xO i' ⇒ Node (ins default i' a l) o r
        | xI i' ⇒ Node l o (ins default i' a r)
        end
    end.

Definition insert {A: Type} (i: positive) (a: A) (t: trie_table
A)
                 : trie_table A :=
  (fst t, ins (fst t) i a (snd t)).

Definition three_ten : trie_table bool :=
 insert 3 true (insert 10 true (empty false)).

Eval compute in three_ten.
(* = (false,
        Node (Node Leaf false (Node (Node Leaf true Leaf) false Leaf))
                  false
                  (Node Leaf true Leaf))
     : trie_table bool  *)

Eval compute in
   map (fun i ⇒ lookup i three_ten) [3;1;4;1;5]%positive.
(*      = true; false; false; false; false  : list bool *)
```

From `N*logN*logN` to `N*logN`

```
Module FastEnough.

Fixpoint loop (input: list positive) (c: nat) (table: trie_table
bool) : nat :=
  match input with
   | nil ⇒ c
   | a::al ⇒ if lookup a table
                  then loop al (1+c) table
                  else loop al c (insert a true table)
 end.

Definition collisions (input: list positive) := loop input 0
(empty false).

Example collisions_pi: collisions [3;1;4;1;5;9;2;6]%positive =
1.
Proof. reflexivity. Qed.

End FastEnough.
```

This program takes `O(N log N)` time: the loop executes `N` iterations, the `lookup` takes `log N` time, the `insert` takes `log N` time. One might worry about `1+c` computed in the natural numbers (unary representation), but this evaluates in one step to `S c`, which takes constant time, no matter how long `c` is. In "real life", one might be advised to use `Z` instead of `nat` for the `c` variables, in which case, `1+c` takes worst-case `log N`, and average-case constant time.

Exercise: 2 stars (successor_of_Z_constant_time)

Explain why the average-case time for successor of a binary integer, with carry, is constant time. Assume that the input integer is random (uniform distribution from 1 to N), or assume that we are iterating successor starting at 1, so that each number from 1 to N is touched exactly once — whichever way you like.

```
(* explain here
*)
```

□

Proving the Correctness of Trie Tables

Trie tables are just another implementation of the `Maps` abstract data type. What we have to prove is the same as usual for an ADT: define a representation invariant, define an abstraction relation, prove that the operations respect the invariant and the abstraction relation.

We will indeed do that. But this time we'll take a different approach. Instead of defining a "natural" abstraction relation based on what we see in the data structure, we'll define an abstraction relation that says, "what you get is what you get." This will work, but it means we've moved the work into directly proving some things about the relation between the `lookup` and the `insert` operators.

Lemmas About the Relation Between `lookup` and `insert`

Exercise: 1 star (look_leaf)

```
Lemma look_leaf:
 ∀ A (a:A) j, look a j Leaf = a.
(* FILL IN HERE *) Admitted.
```

☐

Exercise: 2 stars (look_ins_same)

This is a rather simple induction.

```
Lemma look_ins_same: ∀ {A} a k (v:A) t, look a k (ins a k v t) =
v.
(* FILL IN HERE *) Admitted.
```

☐

Exercise: 3 stars (look_ins_same)

Induction on j? Induction on t? Do you feel lucky?

```
Lemma look_ins_other: ∀ {A} a j k (v:A) t,
   j ≠ k → look a j (ins a k v t) = look a j t.
(* FILL IN HERE *) Admitted.
```

☐

Bijection Between `positive` and `nat`.

In order to relate `lookup` on positives to `total_map` on nats, it's helpful to have a bijection between `positive` and `nat`. We'll relate `1%positive` to `0%nat`, `2%positive` to `1%nat`, and so on.

```
Definition nat2pos (n: nat) : positive := Pos.of_succ_nat n.
Definition pos2nat (n: positive) : nat := pred (Pos.to_nat n).

Lemma pos2nat2pos: ∀ p, nat2pos (pos2nat p) = p.
Proof. (* You don't need to read this proof! *)
intro. unfold nat2pos, pos2nat.
rewrite <- (Pos2Nat.id p) at 2.
destruct (Pos.to_nat p) eqn:?.
pose proof (Pos2Nat.is_pos p). omega.
rewrite <- Pos.of_nat_succ.
reflexivity.
Qed.

Lemma nat2pos2nat: ∀ i, pos2nat (nat2pos i) = i.
Proof. (* You don't need to read this proof! *)
intro. unfold nat2pos, pos2nat.
rewrite SuccNat2Pos.id_succ.
reflexivity.
Qed.
```

Now, use those two lemmas to prove that it's really a bijection!

Exercise: 2 stars (pos2nat_bijective)

```
Lemma pos2nat_injective: ∀ p q, pos2nat p = pos2nat q → p=q.
(* FILL IN HERE *) Admitted.

Lemma nat2pos_injective: ∀ i j, nat2pos i = nat2pos j → i=j.
(* FILL IN HERE *) Admitted.
```
☐

Proving That Tries are a "Table" ADT.

Representation invariant. Under what conditions is a trie well-formed? Fill in the simplest thing you can, to start; then correct it later as necessary.

```
Definition is_trie {A: Type} (t: trie_table A) : Prop
(* REPLACE THIS LINE WITH ":= _your_definition_ ." *). Admitted.
```

Abstraction relation. This is what we mean by, "what you get is what you get." That is, the abstraction of a `trie_table` is the total function, from naturals to `A` values, that you get by running the `lookup` function. Based on this abstraction relation, it'll be trivial to prove `lookup_relate`. But `insert_relate` will NOT be trivial.

```
Definition abstract {A: Type} (t: trie_table A) (n: nat) : A :=
  lookup (nat2pos n) t.

Definition Abs {A: Type} (t: trie_table A) (m: total_map A) :=
  abstract t = m.
```

Exercise: 2 stars (is_trie)

If you picked a *really simple* representation invariant, these should be easy. Later, if you need to change the representation invariant in order to get the `_relate` proofs to work, then you'll need to fix these proofs.

```
Theorem empty_is_trie: ∀ {A} (default: A), is_trie (empty
default).
(* FILL IN HERE *) Admitted.

Theorem insert_is_trie: ∀ {A} i x (t: trie_table A),
   is_trie t → is_trie (insert i x t).
(* FILL IN HERE *) Admitted.
```
☐

Exercise: 2 stars (empty_relate)

Just unfold a bunch of definitions, use `extensionality`, and use one of the lemmas you proved above, in the section "Lemmas about the relation between `lookup` and `insert`."

```
Theorem empty_relate: ∀ {A} (default: A),
    Abs (empty default) (t_empty default).
Proof.
(* FILL IN HERE *) Admitted.
```
☐

Exercise: 2 stars (lookup_relate)

Given the abstraction relation we've chosen, this one should be really simple.

```
Theorem lookup_relate: ∀ {A} i (t: trie_table A) m,
    is_trie t → Abs t m → lookup i t = m (pos2nat i).
(* FILL IN HERE *) Admitted.
```

☐

Exercise: 3 stars (insert_relate)

Given the abstraction relation we've chosen, this one should NOT be simple. However, you've already done the heavy lifting, with the lemmas `look_ins_same` and `look_ins_other`. You will not need induction here. Instead, unfold a bunch of things, use extensionality, and get to a case analysis on whether `pos2nat k =? pos2nat j`. To handle that case analysis, use `bdestruct`. You may also need `pos2nat_injective`.

```
Theorem insert_relate: ∀ {A} k (v: A) t cts,
    is_trie t →
    Abs t cts →
    Abs (insert k v t) (t_update cts (pos2nat k) v).
(* FILL IN HERE *) Admitted.
```

☐

Sanity Check

```
Example Abs_three_ten:
    Abs
       (insert 3 true (insert 10 true (empty false)))
       (t_update (t_update (t_empty false) (pos2nat 10) true)
(pos2nat 3) true).
Proof.
try (apply insert_relate; [hnf; auto | ]).
try (apply insert_relate; [hnf; auto | ]).
try (apply empty_relate).
(* Change this to Qed once you have is_trie defined and working. *)
(* FILL IN HERE *) Admitted.
```

Conclusion

Efficient functional maps with (positive) integer keys are one of the most important data structures in functional programming. They are used for symbol tables in compilers and static analyzers; to represent directed graphs (the mapping from node-ID to edge-list); and (in general) anywhere that an imperative algorithm uses an array or *requires* a mutable pointer.

Therefore, these *tries* on positive numbers are very important in Coq programming. They were introduced by Xavier Leroy and Sandrine Blazy in the CompCert compiler (2006), and are now available in the Coq standard library as the `PositiveMap` module, which implements the `FMaps` interface. The core implementation of `PositiveMap` is just as shown in this chapter, but `FMaps` uses different names for the functions `insert` and `lookup`, and also provides several other operations on maps.