COLOR

GRAPH COLORING

Required reading: Kempe's graph-coloring algorithm , by Andrew W. Appel, 2016.

Suggested reading: Formal Verification of Coalescing Graph-Coloring Register Allocation , by Sandrine Blazy, Benoit Robillard, and Andrew W. Appel. ESOP 2010: 19th European Symposium on Programming, pp. 145-164, March 2010.

Coloring an undirected graph means, assigning a color to each node, so that any two nodes directly connected by an edge have different colors. The *chromatic number* of a graph is the minimum number of colors needed to color the graph. Graph coloring is NP-complete, so there is no polynomial-time algorithm; but we need to do it anyway, for applications such as register allocation in compilers. So therefore we often use incomplete algorithms: ones that work only on certain classes of graphs, or ones that color *most* but not all of the nodes. Those algorithms are often good enough for important applications.

In this chapter we will study Kempe's algorithm for K-coloring a graph. It was invented by Alfred Kempe in 1879, for use in his attempt to prove the four-color theorem (that every planar graph is 4-colorable). His 4-color proof had a bug; but his algorithm continues to be useful: a (major) variation of it was used in the successful 1976 proof of the 4-color theorem, and in 1979 Kempe's algorithm was adapted by Gregory Chaitin for application to register allocation. It is the Kempe-Chaitin algorithm that we'll prove here.

We implement a program to K-color an undirected graph, perhaps leaving some nodes uncolored. In a register-allocation problem, the graph nodes correspond to variables in a program, the colors correspond to registers, and the graph edges are interference constraints: two nodes connected by an edge cannot be assigned the same color. Nodes left uncolored are "spilled," that is, a register allocator would implement such nodes in memory locations instead of in registers. We desire to have as few uncolored nodes as possible, but this desire is not formally specified.

In this exercise we show a simple and unsophisticated algorithm; the program described by Blazy et al. (cited above) is more sophisticated in several ways, such as the use of "register coalescing" to get better results and the use of worklists to make it run faster.

Our algorithm does, at least, make use of efficient data structures for representing undirected graphs, adjacency sets, and so on.

Preliminaries: Representing Graphs

In the Trie chapter we saw how to represent efficient maps (lookup tables) where the keys are `positive` numbers in Coq. Those tries are implemented in the Coq standard library as `FMaps`, functional maps, and we will use them directly from the standard library. `FMaps` represent *partial functions*, that is, mapping keys to `option(t)` for whatever `t`.

We will also use `FSets`, efficient sets of keys; you can *think* of those as FMaps from keys to `unit`, where `None` means absent and `Some tt` means present; but their implementation is a bit more efficient.

```
Require Import List.
Require Import FSets. (* Efficient functional sets *)
Require Import FMaps. (* Efficient functional maps *)
Require Import Perm.
(* to use <? notation and bdestruct tactic *)
```

The nodes in our graph will be named by positive numbers. `FSets` and `FMaps` are interfaces for sets and maps over an element type. One instance is when the element type is `positive`, with a particular comparison operator corresponding to easy lookup in tries. The Coq module for this element type (with its total order) is `PositiveOrderedTypeBits`. We'll use `E` as an abbreviation for this module name.

```
Module E := PositiveOrderedTypeBits.
Print Module E.
Print E.t.
```

The `Module Type FSetInterface.S` gives the API of "functional sets." One instance of this, `PositiveSet`, has keys = positive numbers. We abbreviate this as `Module S`.

```
Module S <: FSetInterface.S := PositiveSet.
Print Module S.
Print S.elt.
```

And similarly for functional maps over positives

```
Module M <: FMapInterface.S := PositiveMap.
Print Module M.
Print M.E.
```

Lemmas About Sets and Maps

In order to reason about a graph coloring algorithm, we need to prove lemmas such as, "if you remove an element (one domain->range binding) from a finite map, then the result is a new finite map whose domain has fewer elements." (Duh!) But to prove this, we need to build up some definitions and lemmas. We start by importing some modules that have some already-proved properties of FMaps.

```
Module WF := WFacts_fun E M.
(* Library of useful lemmas about maps *)
Module WP := WProperties_fun E M.
(* More useful stuff about maps *)
Print Module WF.
Print Module WP.

Check E.lt. (*   : positive -> positive -> Prop *)
```

`E.lt` is a comparison predicate on `positive` numbers. It is *not* the usual less-than operator; it is a different ordering that is more compatible with the order that a Positive Trie arranges its keys. In the application of certain lemmas about maps and sets, we will need the facts that `E.lt` is a `StrictOrder` (irreflexive and transitive) and respects a congruence over equality (is `Proper` for `eq ==> eq ==> iff`). As shown here, we just have to dig up these facts from a submodule of a submodule of a submodule of `M`.

```
Lemma lt_strict: StrictOrder E.lt.
Proof. exact M.ME.MO.IsTO.lt_strorder. Qed.

Lemma lt_proper: Proper (eq ==> eq ==> iff) E.lt.
Proof. exact M.ME.MO.IsTO.lt_compat. Qed.
```

The domain of a map is the set of elements that map to Some(_). To calculate the domain, we can use `M.fold`, an operation that comes with the `FMaps` abstract data type. It takes a map `m`, function `f` and base value `b`, and calculates `f` x_1 y_1 `(f` x_2 y_2 `(f` x_3 y_3 `(... (f xn yn b)...)))`, where `(xi,yi)` are the individual elements of `m`. That is, `M.find xi m = Some yi`, for each `i`.

So, to compute the domain, we just use an `f` function that adds `xi` to a set; mapping this over all the nodes will add all the keys in `m` to the set `S.empty`.

```
Definition Mdomain {A} (m: M.t A) : S.t :=
   M.fold (fun n _ s ⇒ S.add n s) m S.empty.
```

Example: Make a map from *node* (represented as `positive`) to *set of node* (represented as `S.t`), in which nodes `3,9,2` each map to the empty set, and no other nodes map to anything.

```
Definition example_map : M.t S.t :=
(M.add 3%positive S.empty
  (M.add 9%positive S.empty
   (M.add 2%positive S.empty (M.empty S.t )))).
```

```
Example domain_example_map:
    S.elements (Mdomain example_map) = [2;9;3]%positive.
Proof. compute. reflexivity. Qed.
```

equivlistA

```
Print equivlistA. (*
   fun {A : Type} (eqA : A -> A -> Prop) (l l' : list A) =>
            forall x : A, InA eqA x l <-> InA eqA x l'
   : forall {A:Type}, (A->A->Prop) -> list A -> list A -
> Prop *)
```

Suppose two lists `al,bl` both contain the same elements, not necessarily in the same order. That is, `∀x:A, In x al ↔ In x bl`. In fact from this definition you can see that `al` or `bl` might even have different numbers of repetitions of certain elements. Then we say the lists are "equivalent."

We can generalize this. Suppose instead of `In x al`, which says that the value `x` is in the list `al`, we use a different equivalence relation on that `A`. That is, `InA eqA x al` says that some element of `al` is *equivalent* to `x`, using the equivalence relation `eqA`. For example:

```
Definition same_mod_10 (i j: nat) := i mod 10 = j mod 10.
Example InA_example: InA same_mod_10 27 [3;17;2].
Proof. right. left. compute. reflexivity. Qed.
```

The predicate `equivlistA eqA al bl` says that lists `al` and `bl` have equivalent sets of elements, using the equivalence relation `eqA`. For example:

```
Example equivlistA_example: equivlistA same_mod_10 [3; 17] [7;
3; 27].
Proof.
split; intro.
inv H. right; left. auto.
inv H1. left. apply H0.
inv H0.
inv H. right; left. apply H1.
inv H1. left. apply H0.
inv H0. right. left. apply H1.
inv H1.
Qed.
```

SortA_equivlistA_eqlistA

Suppose two lists `al,bl` are "equivalent:" they contain the same set of elements (modulo an equivalence relation `eqA` on elements, perhaps in different orders, and perhaps with different numbers of repetitions). That is, suppose `equivlistA eqA al bl`.

And suppose list `al` is sorted, in some strict total order (respecting the same equivalence relation `eqA`). And suppose list `bl` is sorted. Then the lists must be *equal* (modulo `eqA`).

Just to make this easier to think about, suppose eqA is just ordinary equality. Then if al and bl contain the same set of elements (perhaps reordered), and each list is sorted (by *less-than*, not by *less-or-equal*), then they must be equal. Obviously.

That's what the theorem SortA_equivlistA_eqlistA says, in the Coq library:

```
Check SortA_equivlistA_eqlistA. (*
     : forall (A : Type) (eqA : A -> A -> Prop),
       Equivalence eqA ->
       forall ltA : A -> A -> Prop,
       StrictOrder ltA ->
       Proper (eqA ==> eqA ==> iff) ltA ->
       forall l l' : list A,
       Sorted ltA l ->
       Sorted ltA l' -> equivlistA eqA l l' -
> eqlistA eqA l l'  *)
```

That is, suppose eqA is an equivalence relation on type A, that is, eqA is reflexive, symmetric, and transitive. And suppose ltA is a strict order, that is, irreflexive and transitive. And suppose ltA respects the equivalence relation, that is, if eqA x x' and eqA y y', then ltA x y ↔ ltA x' y'. THEN, if l is sorted (using the comparison ltA), and l' is sorted, and l,l' contain equivalent sets of elements, then l,l' must be equal lists, modulo the equivalence relation.

To make this easier to think about, let's use ordinary equality for eqA. We will be making sets and maps over the "node" type, E.t, but that's just type positive. Therefore, the equivalence E.eq: E.t → E.t → Prop is just the same as eq.

```
Goal E.t = positive. Proof. reflexivity. Qed.
Goal E.eq = @eq positive. Proof. reflexivity. Qed.
```

And therefore, eqlistA E.eq al bl means the same as al=bl.

```
Lemma eqlistA_Eeq_eq: ∀ al bl, eqlistA E.eq al bl ↔ al=bl.
Proof.
split; intro.
* induction H. reflexivity. unfold E.eq in H. subst.
reflexivity.
* subst. induction bl. constructor. constructor.
   unfold E.eq. reflexivity. assumption.
Qed.
```

So now, the theorem: if al and bl are sorted, and contain "the same" elements, then they are equal:

```
Lemma SortE_equivlistE_eqlistE:
 ∀ al bl, Sorted E.lt al →
                   Sorted E.lt bl →
                   equivlistA E.eq al bl → eqlistA E.eq al bl.
Proof.
  apply SortA_equivlistA_eqlistA; auto.
  apply lt_strict.
  apply lt_proper.
Qed.
```

If list l is sorted, and you apply List.filter to remove the elements on which f is false, then the result is still sorted. Obviously.

```
Lemma filter_sortE: ∀ f l,
     Sorted E.lt l → Sorted E.lt (List.filter f l).
Proof.
  apply filter_sort with E.eq; auto.
  apply lt_strict.
  apply lt_proper.
Qed.
```

S.remove and S.elements

The FSets interface (and therefore our Module S) provides these two functions:

```
Check S.remove. (* : S.elt -> S.t -> S.t *)
Check S.elements. (* : S.t -> list S.elt *)
```

In module S, of course, S.elt=positive, as these are sets of positive numbers.

Now, this relationship between S.remove and S.elements will soon be useful:

```
Lemma Sremove_elements: ∀ (i: E.t) (s: S.t),
  S.In i s →
     S.elements (S.remove i s) =
         List.filter (fun x ⇒ if E.eq_dec x i then false else
true) (S.elements s).
Abort.
(* Before we prove that, there is some preliminary work to do. *)
```

That is, if i is in the set s, then the elements of S.remove i s is the list that you get by filtering i out of S.elements s. Go ahead and prove it!

Exercise: 3 stars (Sremove_elements)

```
Lemma Proper_eq_eq:
  ∀ f, Proper (E.eq ==> @eq bool) f.
Proof.
unfold Proper. unfold respectful.
(* FILL IN HERE *) Admitted.

Lemma Sremove_elements: ∀ (i: E.t) (s: S.t),
  S.In i s →
     S.elements (S.remove i s) =
         List.filter (fun x ⇒ if E.eq_dec x i then false else
true) (S.elements s).
Proof.
intros.
apply eqlistA_Eeq_eq.
apply SortE_equivlistE_eqlistE.
* (* To prove this one, SearchAbout S.elements *)
(* FILL IN HERE *) admit.
* (* Use filter_sortE to prove this one *)
(* FILL IN HERE *) admit.
*
intro j.
```

```
rewrite filter_InA; [ | apply Proper_eq_eq].
destruct (E.eq_dec j i).
(* To prove this one, you'll need  S.remove_1, S.remove_2, S.remove_3,
    S.elements_1, S.elements_2. *)
 + (* j=i *)
(* FILL IN HERE *) admit.
 + (* j <> i *)
(* FILL IN HERE *) admit.
(* FILL IN HERE *) Admitted.
```

□

Lists of (key,value) Pairs

The elements of a finite map from positives to type A (that is, the `M.elements` of a `M.t A`) is a list of pairs `(positive*A)`.

```
Check M.elements. (*  : forall A : Type, M.t A -
> list (positive * A) *)
```

Let's start with a little lemma about lists of pairs: Suppose `l: list (positive*A)`. Then `j` is in `map fst l` iff there is some e such that (j,e) is in l.

Exercise: 2 stars (InA_map_fst_key)

```
Lemma InA_map_fst_key:
 ∀ A j l,
 InA E.eq j (map (@fst M.E.t A) l) ↔ ∃ e, InA (@M.eq_key_elt A)
(j,e) l.
(* FILL IN HERE *) Admitted.
```

□

Exercise: 3 stars (Sorted_lt_key)

The function `M.lt_key` compares two elements of an `M.elements` list, that is, two pairs of type `positive*A`, by just comparing their first elements using `E.lt`. Therefore, an elements list (of type `list(positive*A)` is `Sorted` by `M.lt_key` iff its list-of-first-elements is `Sorted` by `E.lt`.

```
Lemma Sorted_lt_key:
  ∀ A (al: list (positive*A)),
   Sorted (@M.lt_key A) al ↔ Sorted E.lt (map (@fst positive A)
al).
Proof.
(* FILL IN HERE *) Admitted.
```

□

Cardinality

The *cardinality* of a set is the number of distinct elements. The cardinality of a finite map is, essentially, the cardinality of its domain set.

Exercise: 4 stars (cardinal_map)

```
Lemma cardinal_map: ∀ A B (f: A → B) g,
     M.cardinal (M.map f g) = M.cardinal g.
```

Hint: To prove this theorem, I used these lemmas. You might find a different way.

```
Check M.cardinal_1.
Check M.elements_1.
Check M.elements_2.
Check M.elements_3.
Check map_length.
Check eqlistA_length.
Check SortE_equivlistE_eqlistE.
Check InA_map_fst_key.
Check WF.map_mapsto_iff.
Check Sorted_lt_key.

(* FILL IN HERE *) Admitted.
```

□

Exercise: 4 stars (Sremove_cardinal_less)

```
Lemma Sremove_cardinal_less: ∀ i s,
        S.In i s → S.cardinal (S.remove i s) < S.cardinal s.
Proof.
intros.
repeat rewrite S.cardinal_1.
generalize (Sremove_elements _ _ H); intro.
rewrite H0; clear H0.
(* FILL IN HERE *) Admitted.
```

□

We have a lemma `SortA_equivlistA_eqlistA` that talks about arbitrary equivalence relations and arbitrary total-order relations (as long as they are compatible. Here is a specialization to a particular equivalence (`M.eq_key_elt`) and order (`M.lt_key`).

```
Lemma specialize_SortA_equivlistA_eqlistA:
  ∀ A al bl,
  Sorted (@M.lt_key A) al →
  Sorted (@M.lt_key A) bl →
  equivlistA (@M.eq_key_elt A) al bl →
  eqlistA (@M.eq_key_elt A) al bl.
Proof.
intros.
apply SortA_equivlistA_eqlistA with (@M.lt_key A); auto.
apply M.eqke_equiv.
apply M.ltk_strorder.
clear.
repeat intro.
unfold M.lt_key, M.eq_key_elt in *.
destruct H, H0. rewrite H,H0. split; auto.
Qed.

Lemma Proper_eq_key_elt:
 ∀ A,
   Proper (@M.eq_key_elt A ==> @M.eq_key_elt A ==> iff)
                (fun x y : E.t * A ⇒ E.lt (fst x) (fst y)).
Proof.
```

```
   repeat intro. destruct H,H0. rewrite H,H0. split; auto.
 Qed.
```

Exercise: 4 stars (Mremove_elements)

```
Lemma Mremove_elements: ∀ A i s,
  M.In i s →
     eqlistA (@M.eq_key_elt A) (M.elements (M.remove i s))
              (List.filter (fun x ⇒ if E.eq_dec (fst x) i then
false else true) (M.elements s)).

(* Hints: *)
Check specialize_SortA_equivlistA_eqlistA.
Check M.elements_1.
Check M.elements_2.
Check M.elements_3.
Check M.remove_1.
Check M.eqke_equiv.
Check M.ltk_strorder.
Check Proper_eq_key_elt.
Check filter_InA.

(* FILL IN HERE *) Admitted.
```

□

Exercise: 3 stars (Mremove_cardinal_less)

```
Lemma Mremove_cardinal_less: ∀ A i (s: M.t A), M.In i s →
        M.cardinal (M.remove i s) < M.cardinal s.
```

Look at the proof of `Sremove_cardinal_less`, if you succeeded in that, for an idea of how to do this one.

```
(* FILL IN HERE *) Admitted.
```

□

Exercise: 2 stars (two_little_lemmas)

```
Lemma fold_right_rev_left:
  ∀ (A B: Type) (f: A → B → A) (l: list B) (i: A),
  fold_left f l i = fold_right (fun x y ⇒ f y x) i (rev l).
(* FILL IN HERE *) Admitted.

Lemma Snot_in_empty: ∀ n, ¬ S.In n S.empty.
(* FILL IN HERE *) Admitted.
```

□

Exercise: 3 stars (Sin_domain)

```
Lemma Sin_domain: ∀ A n (g: M.t A), S.In n (Mdomain g) ↔ M.In n
g.
```

This seems so obvious! But I didn't find a really simple proof of it.

```
(* FILL IN HERE *) Admitted.
```

□

Now Begins the Graph Coloring Program

```
Definition node := E.t.
Definition nodeset := S.t.
Definition nodemap: Type → Type := M.t.
Definition graph := nodemap nodeset.

Definition adj (g: graph) (i: node) : nodeset :=
  match M.find i g with Some a ⇒ a | None ⇒ S.empty end.

Definition undirected (g: graph) :=
   ∀ i j, S.In j (adj g i) → S.In i (adj g j).

Definition no_selfloop (g: graph) := ∀ i, ¬ S.In i (adj g i).

Definition nodes (g: graph) := Mdomain g.

Definition subset_nodes
                    (P: node → nodeset → bool)
                    (g: graph) :=
   M.fold (fun n adj s ⇒ if P n adj then S.add n s else s) g
S.empty.
```

A node has "low degree" if the cardinality of its adjacency set is less than `K`

```
Definition low_deg (K: nat) (n: node) (adj: nodeset) : bool :=
S.cardinal adj <? K.

Definition remove_node (n: node) (g: graph) : graph :=
  M.map (S.remove n) (M.remove n g).
```

Some Proofs in Support of Termination

We need to prove some lemmas related to the termination of the algorithm before we can actually define the `Function`.

Exercise: 3 stars (subset_nodes_sub)

```
Lemma subset_nodes_sub: ∀ P g, S.Subset (subset_nodes P g)
(nodes g).
(* FILL IN HERE *) Admitted.
```

□

Exercise: 3 stars (select_terminates)

```
Lemma select_terminates:
  ∀ (K: nat) (g : graph) (n : S.elt),
   S.choose (subset_nodes (low_deg K) g) = Some n →
   M.cardinal (remove_node n g) < M.cardinal g.
(* FILL IN HERE *) Admitted.
```

□

The Rest of the Algorithm

```
Require Import Recdef.
(* Must import this to use the Function feature *)

Function select (K: nat) (g: graph) {measure M.cardinal g}: list
node :=
  match S.choose (subset_nodes (low_deg K) g) with
  | Some n ⇒ n :: select K (remove_node n g)
  | None ⇒ nil
  end.
Proof. apply select_terminates.
Defined.
(* Do not use Qed on a Function, otherwise it won't Compute! *)

Definition coloring := M.t node.

Definition colors_of (f: coloring) (s: S.t) : S.t :=
   S.fold (fun n s ⇒ match M.find n f with Some c ⇒ S.add c s |
None ⇒ s end) s S.empty.

Definition color1 (palette: S.t) (g: graph) (n: node) (f:
coloring) : coloring :=
   match S.choose (S.diff palette (colors_of f (adj g n))) with
   | Some c ⇒ M.add n c f
   | None ⇒ f
   end.

Definition color (palette: S.t) (g: graph) : coloring :=
  fold_right (color1 palette g) (M.empty _) (select (S.cardinal
palette) g).
```

Proof of Correctness of the Algorithm.

We want to show that any coloring produced by the `color` function actually respects the interference constraints. This property is called `coloring_ok`.

```
Definition coloring_ok (palette: S.t) (g: graph) (f: coloring)
:=
 ∀ i j, S.In j (adj g i) →
     (∀ ci, M.find i f = Some ci → S.In ci palette) ∧
     (∀ ci cj, M.find i f = Some ci → M.find j f = Some cj →
ci≠cj).
```

Exercise: 2 stars (adj_ext)

```
Lemma adj_ext: ∀ g i j, E.eq i j → S.eq (adj g i) (adj g j).
(* FILL IN HERE *) Admitted.
```

□

Exercise: 3 stars (in_colors_of_1)

```
Lemma in_colors_of_1:
  ∀ i s f c, S.In i s → M.find i f = Some c → S.In c (colors_of
f s).
(* FILL IN HERE *) Admitted.
```

□

Exercise: 4 stars (color_correct)

```
Theorem color_correct:
  ∀ palette g,
      no_selfloop g →
      undirected g →
      coloring_ok palette g (color palette g).
(* FILL IN HERE *) Admitted.
```

□

That concludes the proof that the algorithm is correct.

Trying Out the Algorithm on an Actual Test Case

```
Local Open Scope positive.
(* now 1,2,3, etc. are notation for positive numbers. *)
(* Let's use only three colors *)
Definition palette: S.t := fold_right S.add S.empty [1; 2; 3].

Definition add_edge (e: (E.t*E.t)) (g: graph) : graph :=
 M.add (fst e) (S.add (snd e) (adj g (fst e)))
  (M.add (snd e) (S.add (fst e) (adj g (snd e))) g).

Definition mk_graph (el: list (E.t*E.t)) :=
  fold_right add_edge (M.empty _) el.

Definition G :=
    mk_graph [ (5,6); (6,2); (5,2); (1,5); (1,2); (2,4); (1,4)].

Compute (S.elements (Mdomain G)). (* = 4; 2; 6; 1; 5 *)

Compute (M.elements (color palette G)). (* = (4, 1); (2, 3); (6,
2); (1, 2); (5, 1) *)
```

That is our graph coloring: Node 4 is colored with color 1, node 2 with color 3, nodes 6 and 1 with 2, and node 5 with color 1.