```
theory Insert imports Find "$SRC/b_pre_monad/Insert_state" begin

type_synonym ('k,'v,'r) fo = "('k,'v,'r)i12_t"
type_synonym ('k,'v,'r,'leaf,'frame) d (* down_state *) = "('k,'r,'leaf,'frame)find_state*'v"
type_synonym ('k,'v,'r,'frame) u (* up_state *) = "('k,'v,'r)fo*'frame list"

(* insert ---------------------------------------------------------- *)


definition step_down :: "
('k,'r,'frame,'node) frame_ops ⇒
('r,('node,'leaf)dnode,'t) store_ops ⇒
('k,'v,'r,'leaf,'frame) d ⇒ (('k,'v,'r,'leaf,'frame) d,'t) MM" where
"step_down frame_ops store_ops = (
  let find_step =  find_step frame_ops store_ops in
  (% d.
  let (fs,v) = d in
  find_step fs |> fmap (% d'. (d',v)) ))"

(* split_large_leaf:

We have min and max leaf size. We have a large leaf, of size n.

We want to split n so that the left leaf has as many keys as possible.

- Allocate min to right. Remainder is n-min.
- If n-min <= max, allocate to left.
- Else, allocate max to left, and (n-min)-max additional to right.
  - right is therefore n-max



*)

(* Following returns the length of the left leaf *)
definition calculate_leaf_split where
"calculate_leaf_split cs n = (
  let _ = assert_true (n > cs|>max_leaf_size) in
  let left_possibles = n-(cs|>min_leaf_size) in
  case left_possibles ≤ cs|>max_leaf_size of
  True ⇒ left_possibles
  | False ⇒ cs|>max_leaf_size
)"

definition step_bottom :: "
constants ⇒
('k,'v,'leaf) leaf_ops ⇒
('k,'r,'node) node_ops ⇒
('r,('node,'leaf)dnode,'t) store_ops ⇒
('k,'v,'r,'leaf,'frame) d ⇒ (('k,'v,'r,'frame) u + unit,'t) MM" where
"step_bottom cs leaf_ops node_ops store_ops d = (
  let (write,rewrite) = (store_ops|>wrte,store_ops|>rewrite) in
  let (fs,v) = d in
  case dest_F_finished fs of
  None ⇒ (failwith (STR ''insert, step_bottom, 1''))
  | Some(r0,k,r,leaf,stk) ⇒ (
    — ‹ free here? FIXME ›
    let (leaf',_) = (leaf_ops|>leaf_insert) k v leaf in
    let length_leaf' = (leaf_ops|>leaf_length) leaf' in
```

```
    case length_leaf' ≤ cs|>max_leaf_size of
    True ⇒ (
      — ‹ we want to update in place if possible ›
      Disk_leaf leaf' |> rewrite r |> bind (% r'.
      case r' of
      None ⇒
        — ‹ block was updated in place ›
        return (Inr ())
      | Some r' ⇒ return (Inl(I1 r',stk))))
    | False ⇒ (
      let split_point = calculate_leaf_split cs length_leaf' in
      let (leaf1,k',leaf2) = (leaf_ops|>split_large_leaf) split_point leaf' in
      Disk_leaf leaf1 |> write |> bind (% r1.
      Disk_leaf leaf2 |> write |> bind (% r2.
      return (Inl(I2(r1,k',r2),stk)))))))"

(* split_node:

We have min and max node keys.

We need to split a node which has more than max keys (say, n).

We want to split so left node has as many keys as possible.

- Allocate min keys to right node.
- There are n -1 - min keys available for left.
- If n-1-min <= max then allocate to left
- Else, allocate max to left, and n-1-max to right.

*)

definition calculate_node_split where
"calculate_node_split cs n = (
  let _ = assert_true (n > cs|>max_node_keys) in
  let left_possibles = n - 1 - (cs|>min_node_keys) in
  case left_possibles ≤ cs|>max_node_keys of
  True ⇒ left_possibles
  | False ⇒ cs|>max_node_keys
)"

definition step_up :: "
constants ⇒
('k,'r,'node) node_ops ⇒
('k,'r,'frame,'node) frame_ops ⇒
('r,('node,'leaf)dnode,'t) store_ops ⇒
('k,'v,'r,'frame) u ⇒ (('k,'v,'r,'frame) u + unit,'t) MM" where
"step_up cs node_ops frame_ops store_ops u = (
  let (write,rewrite) = (store_ops|>wrte,store_ops|>rewrite) in
  let (fo,stk) = u in
  case stk of
  [] ⇒ failwith (STR ''insert, step_up,1'')
  | frm#stk' ⇒ (
    let backing_r = (frame_ops|>backing_node_blk_ref) frm in
    case fo of
    I1 r ⇒ (
      let (k1,r1,k2) = (frame_ops|>get_focus) frm in
      frm |> (frame_ops|>replace) (k1,r1,[],k2) (k1,r,[],k2)
      |> (frame_ops|>frame_to_node)
      |> Disk_node |> rewrite backing_r |> bind (% r2.
```

```
    case r2 of
    None ⇒ return (Inr ())
    | Some r2 ⇒ return (Inl (I1 r2, stk'))))
  | I2 (r,k,r') ⇒ (
    let (k1,r1,k2) = (frame_ops|>get_focus) frm in
    let n = frm |> (frame_ops|>replace) (k1,r1,[],k2) (k1,r,[(k,r')],k2) |> (frame_ops|>frame_to_node) in
    let n_keys_length = (node_ops|>node_keys_length) n in
    case n_keys_length ≤ (cs|>max_node_keys) of
    True ⇒ (
      Disk_node(n) |> rewrite backing_r |> bind (% r2.
      case r2 of
      None ⇒ return (Inr ())
      | Some r2 ⇒ return (Inl (I1 r2, stk'))))
    | False ⇒ (
      let index = calculate_node_split cs n_keys_length in  (* index counts from 0 *)
      let (n1,k,n2) = (node_ops|>split_node_at_k_index) index n in
      Disk_node(n1) |> write |> bind (% r1.
      Disk_node(n2) |> write |> bind (% r2.
      return (Inl (I2(r1,k,r2),stk'))))) )))"
```

```
definition insert_step :: "
constants ⇒
('k,'v,'leaf) leaf_ops ⇒
('k,'r,'node) node_ops ⇒
('k,'r,'frame,'node) frame_ops ⇒
('r,('node,'leaf)dnode,'t) store_ops ⇒
('k,'v,'r,'leaf,'frame) insert_state ⇒ (('k,'v,'r,'leaf,'frame) insert_state,'t) MM" where
"insert_step cs leaf_ops node_ops frame_ops store_ops = (
 let step_down = step_down frame_ops store_ops in
 let step_bottom = step_bottom cs leaf_ops node_ops store_ops in
 let step_up = step_up cs node_ops frame_ops store_ops in
 let write = store_ops|>wrte in
 (% s.
 case s of
 I_down d ⇒ (
   let (fs,v) = d in
   case dest_F_finished fs of
   None ⇒ (step_down d |> fmap (% d. I_down d))
   | Some _ ⇒ step_bottom d |> bind (% bot.
     case bot of
     Inr () ⇒ return I_finished_with_mutate
     | Inl u ⇒ return (I_up u)))
 | I_up u ⇒ (
   let (fo,stk) = u in
   case stk of
   [] ⇒ (
     case fo of
     I1 r ⇒ return (I_finished r)
     | I2(r1,k,r2) ⇒ (
       (Disk_node((node_ops|>node_make_small_root)(r1,k,r2)) |> write |> bind (% r.
       return (I_finished r)))))
   | _ ⇒ (step_up u |> bind (% u.
     case u of
     Inr () ⇒ return I_finished_with_mutate
     | Inl u ⇒ return (I_up u))))
 | I_finished _ ⇒ (failwith (STR ''insert_step 1''))
 | I_finished_with_mutate ⇒ (failwith (STR ''insert_step 2''))))"
```

```
definition insert_big_step :: "
constants ⇒ 
('k,'v,'leaf) leaf_ops ⇒
('k,'r,'node) node_ops ⇒ 
('k,'r,'frame,'node) frame_ops ⇒ 
('r,('node,'leaf)dnode,'t) store_ops ⇒ 
('k,'v,'r,'leaf,'frame) insert_state ⇒ (('k,'v,'r,'leaf,'frame) insert_state,'t) MM" where
"insert_big_step cs leaf_ops node_ops frame_ops store_ops = (
  let insert_step = insert_step cs leaf_ops node_ops frame_ops store_ops in
  (% i.
  iter_m (% i. case i of
    I_finished r ⇒ (return None)
    | I_finished_with_mutate ⇒ (return None)
    | _ ⇒ (insert_step i |> fmap Some))
    i))"

definition insert :: "
constants ⇒ 
('k,'v,'leaf) leaf_ops ⇒
('k,'r,'node) node_ops ⇒ 
('k,'r,'frame,'node) frame_ops ⇒ 
('r,('node,'leaf)dnode,'t) store_ops ⇒ 
('r ⇒ (bool,'t) MM) ⇒ 
'r ⇒ 'k ⇒ 'v ⇒ ('r option,'t) MM" where
"insert cs leaf_ops node_ops frame_ops store_ops check_tree_at_r'  = (% r k v.
  let i = make_initial_insert_state r k v in
  insert_big_step cs leaf_ops node_ops frame_ops store_ops i |> bind (% i.
  case i of
  I_finished r ⇒ (check_tree_at_r' r |> bind (% _. return (Some r)))
  | I_finished_with_mutate ⇒ (check_tree_at_r' r |> bind (% _. return None))
  | _ ⇒ failwith (STR ''insert 1'')
))"



end






(*
export_code
"Code_Numeral.int_of_integer"
fmap
Disk_node
make_constants
make_store_ops
make_initial_find_state
I1
I_down
insert_step

in OCaml file "/tmp/insert_with_mutation.ml"
*)
```