```
theory Find imports Post_monad "$SRC/b_pre_monad/Find_state" begin

(* find ---------------------------------------------------------- *)

definition find_step :: "
('k,'r,'frame,'node) frame_ops ⇒
('r,('node,'leaf)dnode,'t) store_ops ⇒
('k,'r,'leaf,'frame) find_state ⇒ (('k,'r,'leaf,'frame) find_state,'t) MM" where
"find_step frame_ops store_ops = (
  let read = store_ops|>read in
  (% fs.
  case fs of
  F_finished _ ⇒ (failwith (STR ''find_step 1''))
  | F_down(r0,k,r,stk) ⇒ (
    read r |>fmap (% f.
    case f of
    Disk_node n ⇒ (
      let frm = (frame_ops|>split_node_on_key) r k n in
      let r = (frame_ops|>midpoint) frm in
      F_down(r0,k,r,frm#stk))
    | Disk_leaf leaf ⇒ F_finished(r0,k,r,leaf,stk)))))"

definition find_big_step :: "
('k,'r,'frame,'node) frame_ops ⇒
('r,('node,'leaf)dnode,'t) store_ops ⇒
('k,'r,'leaf,'frame) find_state ⇒ (('k,'r,'leaf,'frame) find_state,'t) MM" where
"find_big_step frame_ops store_ops = (
  let step = find_step frame_ops store_ops in
  (% i.
  iter_m (% i. case i of
    F_finished _ ⇒ (return None)
    | _ ⇒ (step i |> fmap Some))
    i))"

definition find :: "
('k,'r,'frame,'node) frame_ops ⇒
('r,('node,'leaf)dnode,'t) store_ops ⇒
'r ⇒ 'k ⇒ ('r * 'leaf * 'frame list,'t) MM" where
"find frame_ops store_ops r k = (
  let s = make_initial_find_state k r in
  find_big_step frame_ops store_ops s |> bind (% s.
  case s of
  F_finished(r0,k,r,kvs,stk) ⇒ return (r,kvs,stk)
  | _ ⇒ failwith (STR ''find 1'')))"


(* attempt to do the same, but within a locale *)
(*
locale f =
  fixes cs :: "constants" and
  k_cmp :: "'k ord" and
  frame_ops :: "('k,'r,'frame,'left_half,'right_half,'node) frame_ops" and
  store_ops :: "('r,('node,'leaf)dnode,'t) store_ops"


definition (in f) find_step :: "
('k,'r,'leaf,'frame) find_state ⇒ (('k,'r,'leaf,'frame) find_state,'t) MM" where
"find_step = (
```

```
  let read = store_ops|>read in
  (% fs.
  case fs of
  F_finished _ ⇒ (failwith (STR ''find_step 1''))
  | F_down(r0,k,r,stk) ⇒ (
    read r |>fmap (% f.
    case f of
    Disk_node n ⇒ (
      let frm = (frame_ops|>split_node_on_key) n k in
      let r = (frame_ops|>midpoint) frm in
      F_down(r0,k,r,frm#stk))
    | Disk_leaf leaf ⇒ F_finished(r0,k,r,leaf,stk)))))"

definition (in f) find_big_step :: "
('k,'r,'leaf,'frame) find_state ⇒ (('k,'r,'leaf,'frame) find_state,'t) MM" where
"find_big_step = (
  (% i.
  iter_m (% i. case i of
    F_finished _ ⇒ (return None)
    | _ ⇒ (find_step i |> fmap Some))
    i))"


definition (in f) find :: "'r ⇒ 'k ⇒ ('r * 'leaf * 'frame list,'t) MM" where
"find  r k = (
  let s = make_initial_find_state k r in
  find_big_step s |> bind (% s.
  case s of
  F_finished(r0,k,r,kvs,stk) ⇒ return (r,kvs,stk)
  | _ ⇒ failwith (STR ''find 1'')))"

print_locale! f

thm f.find_def

definition find2 :: "
('k,'r,'frame,'left_half,'right_half,'node) frame_ops ⇒
('r,('node,'leaf)dnode,'t) store_ops ⇒
'r ⇒ 'k ⇒ ('r * 'leaf * 'frame list,'t) MM" where
"find2 x y = f.find x y"

lemma find_def_2: "find2 x y r k = undefined"
  apply(simp add: find2_def)
  apply(simp cong: find_state.case_cong add: f.find_def f.find_big_step_def f.find_step_def)
  oops

lemma find_def_2[code]: "find2 x y r k =
(let s = make_initial_find_state k r
     in (case s of
         F_down prod' ⇒
           (case prod' of
            (r0, k, r, stk) ⇒
              (y |> read) r |>
              fmap
               (case_dnode
                 (λn. let frm = (x |> split_node_on_key) n k in F_down (r0, k, (x |> midpoint) frm, frm # stk))
                 (λleaf. F_finished (r0, k, r, leaf, stk)))) |>
           fmap Some
         | F_finished x ⇒ return None) |>
```

```
        bind
         (case_option (return s)
           (iter_m
             (case_find_state
               (λprod.
                   (case prod of
                    (r0, k, r, stk) ⇒
                      (y |> read) r |>
                      fmap
                       (case_dnode
                         (λn. let frm = (x |> split_node_on_key) n k
                               in F_down (r0, k, (x |> midpoint) frm, frm # stk))
                         (λleaf. F_finished (r0, k, r, leaf, stk)))) |>
                   fmap Some)
               (λx. return None)))) |>
        bind (λs. case s of F_down prod' ⇒ failwith STR ''find 1'' | F_finished (r0, k, ba) ⇒ return ba))"
  apply(simp add: find2_def)
  apply(simp cong: find_state.case_cong add: f.find_def f.find_big_step_def f.find_step_def)
  done
*)

end

(* old ============================================================



(* find_trans *)

(*
definition find_trans :: "(store * fs) trans_t" where
"find_trans = { ((s,fs),(s',fs')). ( s|>(find_step fs|>dest_M) = (s',Ok fs')) }"
*)

(* lemmas *)

(* wf_fts is invariant *)
(*
definition invariant_lem :: "bool" where
"invariant_lem = (
  ! P s t0.
  ((% s_fs. let (_,fs) =  s_fs in wellformed_find_state s t0 fs) = P) ⟶ invariant find_trans P)"
*)

*)
```