```
theory Delete imports Find "$SRC/b_pre_monad/Delete_state" begin

(* FIXME merge in documentation from Delete *)

(* NOTE these are repeated from Delete_state, because otherwise they are shadowed by eg insert.fo *)
type_synonym ('r,'node,'leaf)fo = "('r,'node,'leaf) del_t"  (* focus *)
type_synonym ('r,'node,'leaf,'frame)u = "('r,'node,'leaf)fo * 'frame list"
type_synonym ('k,'r,'leaf,'frame)d = "('k,'r,'leaf,'frame)find_state * 'r"

(* node steal ------------------------------------------------------ *)

(* args are left split node context, focus, right sib; returns updated parent

FIXME maybe it makes more sense to deal with the context in isolation, and return r*k*r

NOTE rs,ks as args for node_steal_xxx
 *)

(* FIXME we also want a version that mutates in place *)

(* delete ------------------------------------------------------------  *)

(* after a merge, the parent may become "small", or even have no keys at all if there
was only one key to begin with; this operation tags a node that is small, or even has no
keys at all *)
definition post_merge ::
  "constants ⇒
('k,'r,'node) node_ops ⇒
('r,('node,'leaf)dnode,'t)store_ops ⇒
'node ⇒ (('r,'node,'leaf)fo,'t)MM"
where
"post_merge cs node_ops store_ops n = (
  case ((node_ops|>node_keys_length)n) < cs|>min_node_keys of
  True ⇒ (return (D_small_node(n)))
  | False ⇒ (
    Disk_node(n)|>(store_ops|>wrte)|>bind (% r.
    return (D_updated_subtree(r)))))"

definition step_up_small_leaf where
"step_up_small_leaf  cs leaf_ops node_ops frame_ops store_ops frm leaf = (
  let (read,write) = (store_ops|>read,store_ops|>wrte) in
  let post_merge = post_merge cs node_ops store_ops in
    — ‹NOTE stack is not empty, so at least one sibling; then a small leaf is expected to have FIXME minleafsize-1 entries›
    let _ = (frame_ops|>dbg_frame) frm in
    case (frame_ops|>get_focus_and_right_sibling) frm of
    None ⇒ (
      — ‹ steal or merge from left ›
      case (frame_ops|>get_left_sibling_and_focus) frm of
      None ⇒ failwith (STR ''impossible'')
      | Some (k1,r1,k2,r2,k3) ⇒ (
      r1 |> read |> fmap dest_Disk_leaf |> bind (% left_leaf.
      case (leaf_ops|>leaf_length) left_leaf = cs|>min_leaf_size of
      True ⇒ (
        — ‹ merge from left ›
        (leaf_ops|>leaf_merge) (left_leaf,leaf) |> (% leaf.
        write (Disk_leaf(leaf)) |> bind (% r.
        frm |> (frame_ops|>replace) (k1,r1,[(k2,r2)],k3) (k1,r,[],k3)
        |> (frame_ops|>frame_to_node) |> post_merge)))
      | False ⇒ (
        — ‹ steal from left ›
```

```
    (leaf_ops|>leaf_steal_left) (left_leaf,leaf) |> (% (left_leaf,k',leaf).
    write (Disk_leaf(left_leaf)) |> bind (% r1'.
    write (Disk_leaf(leaf)) |> bind (% r2'.
    frm |> (frame_ops|>replace) (k1,r1,[(k2,r2)],k3) (k1,r1',[(k',r2')],k3)
    |> (frame_ops|>frame_to_node) |> Disk_node
    |> write |> fmap D_updated_subtree)))))))
  | Some (k1,r1,k2,r2,k3) ⇒ (
   — ‹ steal or merge from right ›
   r2 |> read |> fmap dest_Disk_leaf |> bind (% right_leaf.
   case (leaf_ops|>leaf_length) right_leaf = cs|>min_leaf_size of
   True ⇒ (
    — ‹ merge from right ›
    (leaf_ops|>leaf_merge) (leaf,right_leaf) |> (% leaf.
    write (Disk_leaf(leaf)) |> bind (% r.
    frm |> (frame_ops|>replace) (k1,r1,[(k2,r2)],k3) (k1,r,[],k3)
    |> (frame_ops|>frame_to_node) |> post_merge)))
   | False ⇒ (
    — ‹ steal from right ›
    (leaf_ops|>leaf_steal_right) (leaf,right_leaf) |> (% (leaf,k',right_leaf).
    write (Disk_leaf(leaf)) |> bind (% r1'.
    write (Disk_leaf(right_leaf)) |> bind (% r2'.
    frm |> (frame_ops|>replace) (k1,r1,[(k2,r2)],k3) (k1,r1',[(k',r2')],k3)
    |> (frame_ops|>frame_to_node) |> Disk_node
    |> write |> fmap D_updated_subtree)))))))
"

definition step_up_small_node where
"step_up_small_node cs (leaf_ops::('k,'v,'leaf)leaf_ops) node_ops frame_ops store_ops frm n = (
 let (read,write) = (store_ops|>read,store_ops|>wrte) in
 let post_merge = post_merge cs node_ops store_ops in
  case (frame_ops|>get_focus_and_right_sibling) frm of
  None ⇒ (
   — ‹ steal or merge from left ›
   case (frame_ops|>get_left_sibling_and_focus) frm of
   None ⇒ failwith (STR ''impossible'')
   | Some (k1,r1,k2,r2,k3) ⇒ (
   r1 |> read |> fmap dest_Disk_node |> bind (% left_sibling.
   case (node_ops|>node_keys_length) left_sibling = cs|>min_node_keys of
   True ⇒ (
    — ‹ merge from left ›
    (node_ops|>node_merge) (left_sibling,k2,n) |> (% n.
    write (Disk_node(n)) |> bind (% r.
    frm |> (frame_ops|>replace) (k1,r1,[(k2,r2)],k3) (k1,r,[],k3)
    |> (frame_ops|>frame_to_node) |> post_merge)))
   | False ⇒ (
    — ‹ steal from left ›
    (node_ops|>node_steal_left) (left_sibling,k2,n) |> (% (left_sibling,k2',n).
    write (Disk_node(left_sibling)) |> bind (% r1'.
    write (Disk_node(n)) |> bind (% r2'.
    frm |> (frame_ops |> replace) (k1,r1,[(k2,r2)],k3) (k1,r1',[(k2',r2')],k3)
    |> (frame_ops|>frame_to_node) |> Disk_node
    |> write |> fmap D_updated_subtree)))))))
  | Some (k1,r1,k2,r2,k3) ⇒ (
   — ‹ steal or merge from right ›
   r2 |> read |> fmap dest_Disk_node |> bind (% right_sibling.
   case (node_ops|>node_keys_length) right_sibling = cs|>min_node_keys of
   True ⇒ (
    — ‹ merge from right ›
    (node_ops|>node_merge) (n,k2,right_sibling) |> (% n.
    write (Disk_node(n)) |> bind (% r.
    frm |> (frame_ops|>replace) (k1,r1,[(k2,r2)],k3) (k1,r,[],k3)
```

```
        |> (frame_ops|>frame_to_node) |> post_merge)))
      | False ⇒ (
        — ‹ steal from right ›
        (node_ops|>node_steal_right) (n,k2,right_sibling) |> (% (n,k2',right_sibling).
        write (Disk_node(n)) |> bind (% r1'.
        write (Disk_node(right_sibling)) |> bind (% r2'.
        frm |> (frame_ops|>replace) (k1,r1,[(k2,r2)],k3) (k1,r1',[(k2',r2')],k3)
        |> (frame_ops|>frame_to_node) |> Disk_node
        |> write |> fmap D_updated_subtree)))))))
"

definition step_up :: "
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,'node,'leaf,'frame)u ⇒ (('r,'node,'leaf,'frame)u,'t) MM" where
"step_up cs leaf_ops node_ops frame_ops store_ops du = (
  let (f,stk) = du in
  let (read,write) = (store_ops|>read,store_ops|>wrte) in
  let post_merge = post_merge cs node_ops store_ops in
  case stk of [] ⇒ (failwith (STR ''delete, step_up'')) | frm#stk' ⇒ (
  let _ = (frame_ops|>dbg_frame) frm in
  — ‹ NOTE p is the parent ›
  — ‹ take the result of what follows, and add the stk' component ›
  (% x. x |> fmap (% y. (y,stk'))) (case f of
  D_updated_subtree r ⇒ (
    frm |> (frame_ops|>get_focus) |> ( % (k1,r1,k2).
    frm |> (frame_ops|>replace) (k1,r1,[],k2) (k1,r,[],k2)
    |> (frame_ops|>frame_to_node) |> Disk_node
    |> write |> fmap D_updated_subtree))
  | D_small_leaf(leaf) ⇒ (
    step_up_small_leaf  cs leaf_ops node_ops frame_ops store_ops frm leaf)
  | D_small_node(n) ⇒ (
    step_up_small_node cs leaf_ops node_ops frame_ops store_ops frm n))))
"


definition delete_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,'node,'frame)delete_state ⇒ (('k,'v,'r,'leaf,'node,'frame)delete_state,'t) MM"
where
"delete_step cs leaf_ops node_ops frame_ops store_ops = (
  let write = store_ops|>wrte in
  (% s.
  case s of
  D_down(f,r0) ⇒ (
    case dest_F_finished f of
    None ⇒ (find_step frame_ops store_ops f |> fmap (% f'. D_down(f',r0)))
    | Some x ⇒ (
      let (r0,k,r,leaf,stk) = x in
      let vopt :: 'v option = (leaf_ops|>leaf_lookup) k leaf in
      case vopt of
      Some _ ⇒ (
        let leaf' = (leaf_ops|>leaf_remove) k leaf in
        case (leaf_ops|>leaf_length) leaf' < cs|>min_leaf_size of
```

```
        True ⇒ (return (D_up(D_small_leaf(leaf'),stk,r0)))
        | False ⇒ (Disk_leaf(leaf') |> write
          |> fmap (% r. D_up(D_updated_subtree(r),stk,r0))))
      | None ⇒ (return (D_finished r0) )))
  | D_up(f,stk,r0) ⇒ (
    case is_Nil' stk of
    True ⇒ (
      case f of
      D_small_leaf leaf ⇒ (Disk_leaf(leaf)|>write|>fmap D_finished)
      | D_small_node(n) ⇒ (
        case (node_ops|>node_keys_length) n = 0 of
        True ⇒ return (D_finished ((node_ops|>node_get_single_r) n))
        | False ⇒ (Disk_node(n)|>write|>fmap D_finished))
      | D_updated_subtree(r) ⇒ (return (D_finished r)))
    | False ⇒ (step_up cs leaf_ops node_ops frame_ops store_ops (f,stk) |> fmap (% (f,stk). D_up(f,stk,r0))))
  | D_finished(r) ⇒ (failwith (STR ''delete_step 1''))))"


definition delete_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,'node,'frame) delete_state ⇒ (('k,'v,'r,'leaf,'node,'frame) delete_state,'t) MM" where
"delete_big_step cs leaf_ops node_ops frame_ops store_ops = (
  let delete_step = delete_step cs leaf_ops node_ops frame_ops store_ops in
  (% d.
  iter_m (% d. case d of
    D_finished _ ⇒ return None
    | _ ⇒ (delete_step d |> fmap Some))
    d))"


definition delete :: "
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  ⇒ ('r,'t) MM" where
"delete cs leaf_ops node_ops frame_ops store_ops check_tree_at_r' = (% r k.
  let d = make_initial_delete_state r k in
  delete_big_step cs leaf_ops node_ops frame_ops store_ops d |> bind (% d.
  case d of
  D_finished r ⇒ (check_tree_at_r' r |> bind (% _. return r))
  | _ ⇒ (failwith (STR ''delete, impossible''))))"

end
```