```
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
 *)


(* 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)))))"

(* FIXME really don't like all the parameterization ... *)
definition step_up :: "
constants ⇒
('k,'v,'leaf) leaf_ops ⇒
('k,'r,'node) node_ops ⇒
('k,'r,'frame,'left_half,'right_half,'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,wrte) = (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 (lh,rh) = ((frame_ops|>left_half)frm,(frame_ops|>right_half)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 ⇒ (
    let (lh,rh) = ((frame_ops|>left_half)frm,(frame_ops|>right_half)frm) in
    (frame_ops|>unsplit) (lh, R(r), rh) |> Disk_node |> write |> fmap D_updated_subtree)
  | D_small_leaf(leaf) ⇒ (
    — ‹NOTE stack is not empty, so at least one sibling; then a small leaf is expected to have min_leaf_size-1 entries›
    let no_right_sibling = is_None ((frame_ops|>rh_dest_cons) rh) in
    case no_right_sibling of
    True ⇒ (
      — ‹steal or merge from left›
      let _ = check_true (% _. ¬ (is_None ((frame_ops|>lh_dest_snoc) lh))) in
      case (frame_ops|>lh_dest_snoc) lh |> dest_Some of
      (lh,r1,_) ⇒
      r1 |> read |> fmap dest_Disk_leaf |> bind (% left_leaf.
      case (leaf_ops|>leaf_length) left_leaf = cs|>min_leaf_size of
      True ⇒ (
        (leaf_ops|>leaf_merge) (left_leaf,leaf) |> (% leaf.
        write (Disk leaf(leaf)) |> bind (% r.
        (frame_ops|>unsplit) (lh,R(r),rh) |> post_merge)))
      | False ⇒ (
        (leaf_ops|>leaf_steal_left) (left_leaf,leaf) |> (% (left_leaf,k1,leaf).
        write (Disk leaf(left_leaf)) |> bind (% r1.
        write (Disk leaf(leaf)) |> bind (% r2.
        (frame_ops|>unsplit) (lh,Rkr(r1,k1,r2),rh) |> Disk_node
        |> write |> fmap D_updated_subtree))))))
    | False ⇒ (
      — ‹steal or merge from right›
      let _ = check true (% _. ¬ (is_None ((frame_ops|>rh_dest_cons) rh))) in
      case (frame_ops|>rh_dest_cons) rh |> dest_Some of
      (_,r1,rh) ⇒
      r1 |> read |> fmap dest_Disk_leaf |> bind (% right_leaf.
      case (leaf_ops|>leaf_length) right_leaf = cs|>min_leaf_size of
      True ⇒ (
        (leaf_ops|>leaf_merge) (leaf,right_leaf) |> (% leaf.
        write (Disk leaf(leaf)) |> bind (% r.
        (frame_ops|>unsplit) (lh,R(r),rh)|> post_merge)))
      | False ⇒ (
        (leaf_ops|>leaf_steal_right) (leaf,right_leaf) |> (% (leaf,k1,right_leaf).
        write (Disk leaf(leaf)) |> bind (% r1.
        write (Disk leaf(right_leaf)) |> bind (% r2.
        (frame_ops|>unsplit) (lh,Rkr(r1,k1,r2),rh) |> Disk_node
        |> write |> fmap D_updated_subtree))))))
  | D_small_node(n) ⇒ (
    let no_right_sibling = is_None ((frame_ops|>rh_dest_cons) rh) in
    case no_right_sibling of
    True ⇒ (
      — ‹steal or merge from left›
      let _ = check_true (% _. ¬ (is_None ((frame_ops|>lh_dest_snoc) lh))) in
      case (frame_ops|>lh_dest_snoc) lh |> dest_Some of
      (lh,r1,k1) ⇒
      r1 |> read |> fmap dest_Disk_node |> bind (% left_sibling.
      case (node_ops|>node_keys_length) left_sibling = cs|>min_node_keys of
      True ⇒ (
        (node_ops|>node_merge) (left_sibling,k1,n) |> (% n.
        write (Disk_node(n)) |> bind (% r.
        (frame_ops|>unsplit) (lh,R(r),rh) |> post_merge)))
      | False ⇒ (
        (node_ops|>node_steal_left) (left_sibling,k1,n) |> (% (left_sibling,k1,n).
        write (Disk_node(left_sibling)) |> bind (% r1.
        write (Disk_node(n)) |> bind (% r2.
        (frame_ops|>unsplit) (lh,Rkr(r1,k1,r2),rh) |>Disk_node
        |> write |> fmap D_updated_subtree))))))
    | False ⇒ (
      — ‹steal or merge from right›
      case (frame_ops|>rh_dest_cons) rh |> dest_Some of
      (k1,r1,rh) ⇒
      r1 |> read |> fmap dest Disk node |> bind (% right sibling.
      case (node_ops|>node_keys_length) right_sibling = cs|>min_node_keys of
      True ⇒ (
        (node_ops|>node_merge) (n,k1,right_sibling) |> (% n.
        write (Disk_node(n)) |> bind (% r.
        (frame ops|>unsplit) (lh,R(r),rh) |> post_merge)))
      | False ⇒ (
        (node_ops|>node_steal_right) (n,k1,right_sibling) |> (% (n,k1,right_sibling).
        write (Disk_node(n)) |> bind (% r1.
        write (Disk_node(right_sibling)) |> bind (% r2.
        (frame ops|>unsplit) (lh,Rkr(r1,k1,r2),rh) |> Disk_node
        |> write |> fmap D_updated_subtree))))))
)))"


definition delete_step :: "
constants ⇒ 'k ord ⇒
 ('k,'v,'leaf) leaf_ops ⇒
('k,'r,'node) node_ops ⇒
('k,'r,'frame,'left_half,'right_half,'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 k_cmp leaf_ops node_ops frame_ops store_ops = (
  let write = store_ops|>wrte in
  let disk_leaf = % kvs. Disk_leaf ((leaf_ops|>mk_leaf) kvs) in
  (% s.
  case s of
  D_down(f,r0) ⇒ (
    case dest_F_finished f of
    None ⇒ (find_step cs k_cmp frame_ops store_ops f |> fmap (% f'. D_down(f',r0)))
    | Some x ⇒ (
      let (r0,k,r,leaf,stk) = x in
      let r0 :: 'r = r0 in
      let kvs = (leaf_ops|>leaf_kvs) leaf in
      let something to delete = (? x : set (kvs|>List.map fst). key_eq k_cmp x k) in
      case something_to_delete of
      True ⇒ (
        let kvs' = kvs|>List.filter (% x. ¬ (key_eq k_cmp (fst x) k)) in
        case List.length kvs' < cs|>min_leaf_size of
        True ⇒ (return (D up(D small leaf((leaf_ops|>mk_leaf)kvs'),stk,undefined)))
        | False ⇒ (disk_leaf(kvs') |> write
          |> fmap (% r. D_up(D_updated_subtree(r),stk,r0))))
      | False ⇒ (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 ord ⇒
('k,'v,'leaf) leaf_ops ⇒
('k,'r,'node) node_ops ⇒
('k,'r,'frame,'left half,'right half,'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 k cmp leaf ops node ops frame ops store ops = (
  let delete_step = delete_step cs k_cmp 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 ord ⇒
('k,'v,'leaf) leaf ops ⇒
('k,'r,'node) node_ops ⇒
('node ⇒ 'r s) ⇒
('node ⇒ 'k s) ⇒
('k,'r,'frame,'left_half,'right_half,'node) frame_ops ⇒
('r,('node,'leaf)dnode,'t)store_ops ⇒
'r ⇒ 'k  ⇒ ('r,'t) MM" where
"delete cs k_cmp leaf_ops node_ops node2rs node2ks frame_ops store_ops r k = (
  let check tree at r = check tree at r r cs k_cmp leaf_ops node_ops node2rs node2ks store_ops in
  let d = make_initial_delete_state r k in
  delete_big_step cs k_cmp 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
```