```
The "Wood Cutting" Problem
-------------------------

Parameters
----------
L ∈ ℕ // length of beam
global_intervals ∈ (ℕ\times ℕ)* // intervals of the final beam in which having a cut is banne
d ""
local_danger ∈ ℕ // min distance from two beams in consecutive layers
n_layers ∈ ℕ // number of layers per beam

Objects
-------

// Board has a length length and it's necessary
// to cut at least once in each cut_interval
Board:= length:ℕ × cut_intervals:(ℕ × ℕ)*

// A wooden segment represented by its length
Piece:= length:ℕ

Option[T] ≔ T ∪ {⊥} ( ⊥ ∉ T)

Functions
---------

// Returns the first element in a list
first: T* →p T
first(seq)
  requires length(seq) ≠ 0;
= seq[0]

// Returns the given list without the first element
rest: T* →p T*
rest(seq)
  requires length(seq) ≠ 0;
= λi ∈ domain(seq)\{...}. seq[i+1]




--------------------
Optimization problem
--------------------
 Given: initial_state ∈ InitialState
 Find: all_decisions ∈ AllDecisions where

   all_decisions = argmin_(all_decisions ∈ AllDecisions).
        choose cost ∈ Cost.
        Machine(all_decisions, initial_state, cost)



-------
```

```
Machine
-------

InitialBoards    ReorderedBoards    CutPieces      FilteredPieces
    -> [BoardReordering] -> [Cutting] -> [FilterPieces] ->  [Stack]
                                                |
                                                v
                                              Waste

Machine ⊂ AllDecisions × InitialState × Cost
AllDecisions := reorder_decisions = DecisionReorderBoards* ×
                cut_decisions = DecisionCutBoard* ×
                filter_decisions = DecisionFilter*
InitialState := initial_boards = Board* ×
                buffer = Board
                reordered_boards = Board* ×
                cut_pieces = Piece* ×
                filtered_pieces = Piece*
Cost := ℕ

// If True, then when performing all_decisions in initial_state
// we arrive to a valid state of the machine (satisfying CorrectBeam)
// with cost total_cost
Machine(all_decisions, initial_state, total_cost) ⇔

  let initial_reordering_state = (
      before = initial_state.initial_boards,
      after = initial_state.reordered_boards,
      buffer = initial_state.buffer
    )
  ∃ final_reordering_state with
    AllReordering(all_decisions.reorder_decisions,
                  initial_reordering_state,
                  final_reordering_state)

  ∧

  let initial_cut_state = (
      before = final_reordering_state.after,
      after = initial_state.cut_pieces
    )
  ∃ final_cut_state with
    AllCut(all_decisions.cut_decisions,
           initial_cut_state,
           final_cut_state)

  ∧

  let initial_filter_state = (
      before = final_cut_state.after,
      after = initial_state.filtered_pieces,
      cost = 0
    )
  ∃ final_filter_state with
```

```
    AllFilter(all_decisions.filter_decisions,
              initial_filter_state,
              final_filter_state)

  ∧

  CorrectBeam(final_filter_state.after)

  ∧

  total_cost = final_filter_state.cost





-------------
Reorder Boards
-------------

// Verifies that reorder_decisions applied to initial_state
// results in final_state.

AllReordering ⊂ DecisionReorderBoards × StateReorderBoards × StateReorderBoards
// If True, when performing reorder_decisions to initial_state,
// we reach final_state
AllReordering(reorder_decisions, initial_state, final_state) ⇔
  let n_ord = len(reorder_decisions)
  ∃ list_initial_board in (Board*)* (
    with len(list_initial_board) = n_ord
    with list_initial_board[0] = initial_state.before
    with list_initial_board[n_ord - 1] = final_state.before
    ) ∧
  ∃ list_ordered_board in (Board*)* (
    with len(list_ordered_board) = n_cut
    with list_ordered_board[0] = initial_state.after
    with list_ordered_board[n_ord - 1] = final_state.after
    )  ∧
  ∃ list_buffer in (Board*)* (
    with len(list_buffer) = n_cut
    with list_buffer[0] = initial_state.buffer
    with list_buffer[n_ord - 1] = final_state.buffer
    )  ∧
  ∀ i ∈ ℕ>0 with i < n_ord (
    let prior_state = (
      before = list_initial_board[i],
      after = list_ordered_board[i],
      buffer = list_buffer[i]
    )
    let latter_state = (
      before = list_initial_board[i+1],
      after = list_ordered_board[i+1],
```

```
      buffer = list_buffer[i+1]
    )
    Reorder(
      reorder_decisions[i],
      prior_state,
      latter_state
    )
  )

Single Reorder
--------------

StateReorderBoards ≔ before:Board* × buffer: Option[Board] × after:Board*
DecisionReorderBoards ≔ { forward, swap }


Reorder ⊆ DecisionReorderBoards × StateReorderBoards × StateReorderBoards

// Moves first element of initial_state.before to final_state.after
Reorder(forward, initial_state, final_state) ⇔
  initial_state.before ≠ [] ∧
  let board = first(initial_state.before) in
  let rest =  rest(initial_state.before) in
  final_state = initial_state with .before = rest
                              with .after = initial_state.after ∘ [board]

// Moves first element of initial_state.before to final_state.buffer
// moving initial_state.buffer to final_state.after if it wasn't empty
Reorder(swap, initial_state, final_state) ⇔
  initial_state.before ≠ [] ∧
  let board = first(initial_state.before) in
  let rest =  rest(initial_state.before) in
  (
    buffer = ⊥ ∧
    final_state = initial_state with .before = rest
                                with .buffer = board
    )
  ∨
  (
    buffer ≠ ⊥ ∧
    final_state = initial_state with .before = rest
                                with .buffer = board
                                with .after initial_state.after ∘ [buffer]
    )






-------
Cutting
-------
```

```
AllCut ⊂ DecisionCutBoard × StateCut × StateCut

// Verifies that cut_decisions applied to initial_state
// results in final_state.
AllCut(cut_decisions, initial_state, final_state) ⇔
  let n_cut = len(cut_decisions)
  ∃ list_ordered_board in (Board*)* (
    with len(list_ordered_board) = n_cut
    with list_ordered_board[0] = initial_state.before
    with list_ordered_board[n_cut - 1] = final_state.before
    )

  ∧

  ∃ list_cut_pieces in (Piece*)* (
    with len(list_cut_pieces) = n_cut
    with list_cut_pieces[0] = initial_state.after
    with list_cut_pieces[n_cut - 1] = final_state.after
    )

  ∧

  ∀ i ∈ ℕ>0 with i < n_cut (
    let prior_state = (
      before = list_ordered_board[i],
      after = list_cut_pieces[i]
    )
    let latter_state = (
      before = list_ordered_board[i+1],
      after = list_cut_pieces[i+1]
    )
    Cut(
      cut_decisions[i],
      prior_state,
      latter_state
    )
  )


Single Cut
----------

Cut ⊂ DecisionCutBoard* × StateCut × StateCut
StateCut:= before: Board* × after: Piece*
DecisionCutBoard:= ℕ*
// A list of cut positions in the board

// Takes the first board from initial_state.before
// and add the pieces remaining after making the cuts in cut_list
// to final_state.after
Cut(cut_list, initial_state, final_state) ⇔
  initial_state.before ≠ [] ∧
  let board = first(initial_state.before)
  let rest = rest(initial_state.before)
```

```
  // Check no cuts are longer than the board
  ∀ cut in cut_list, cut < board.length

  ∧

  // Check that necessary cuts are performed
  ∀ interval in board.cut_intervals
     ∃ cut in cut_list with interval.0 <= cut <= interval.1

  ∧

  // Define pieces list
  let n = len(cut_list)
  ∃ pieces in Piece* with len(pieces) = n+1

  ∧

  let final_cut_list = [0] ∘ cut_list
  ∀ i in ℕ with i < n
    let start = final_cut_list[i]
    let end = final_cut_list[i+1]
    pieces[i] = end - start

  ∧

  // Include final piece
  pieces[n] = board.length - final_cut_list[n-1] ∧
  let final_state = initial_state
    with .before = rest
    with .after = initial_state.after ∘ pieces





---------
Filtering
---------

AllFilter ⊂ DecisionFilter × StateFilter × StateFilter


// Verifies that filter_decisions applied to initial_state
// results in final_state.
AllFilter(filter_decisions, initial_state, final_state) ⇔
  let n_fil = len(filter_decisions)
  ∃ list_cut_pieces in (Piece*)* (
    with len(list_cut_pieces) = n_fil
    with list_cut_pieces[0] = initial_state.before
    with list_cut_pieces[n_fil - 1] = final_state.before
    )
```

```
  ∧

  ∃ list_filtered_pieces in (Piece*)* (
    with len(list_filtered_pieces) = n_fil
    with list_filtered_pieces[0] = initial_state.after
    with list_filtered_pieces[n_fil - 1] = final_state.after
    )

  ∧

  ∃ list_cost in ℕ* (
    with len(list_cost) = n_fil
    with list_cost[0] = initial_state.cost
    with list_cost[n_fil - 1] = final_state.cost
    )

  ∧

  ∀ i ∈ ℕ>0 with i < n_fil (
    let prior_state = (
      before = list_cut_pieces[i],
      after = list_filtered_pieces[i],
      cost = list_cost[i]
    )
    let latter_state = (
      before = list_cut_pieces[i+1],
      after = list_filtered_pieces[i+1],
      cost = list_cost[i+1]
    )
    Filter(
      filter_decisions[i],
      prior_state,
      latter_state
    )
  )

Single Filter
-------------

Filter ⊂ DecisionFilter × StateFilter × StateFilter
StateFilter:= before:Piece* × after:Piece* × cost:ℕ
// cost accumulates the lengths of the pieces discarded during filtering
DecisionFilter:= {keep, discard}

// Moves the first element of initial_state.before
// to final_state.after
Filter(keep, initial_state, final_state) ⇔
  initial_state.before ≠ [] ∧
  let piece = first(initial_state.before) in
  let rest = rest(initial_state.before) in
  final_state = initial_state
    with .before = rest
    with .after initial_state.after ∘ [piece]
```

```
// Deletes the first element of initial_state.before
// adding it's length to the cost
Filter(discard, initial_state, final_state)⇔
  initial_state.before ≠ [] ∧
  let new_cost = first(initial_state.before)
  let rest = rest(initial_state.before) in
  final_state = initial_state
    with .before = rest
    with .cost = initial_state.cost + new_cost


--------------------------------------


CorrectBeam \subset State

// Checks that the list of pieces form a beam
// under the desired conditions
CorrectBeam(piece_list) ⇔
  let l = len(piece_list)
  let total_length = ∑j=0..l-1 piece_list[j]
  let h = floor(total_length/L) // number of fully completed layers
  ∃ n in ℕ*
    with len(n) = h + 2
    with n[0]=0
    with n[h+1] = l

  ∧

  ∀ i ≤ h
    // Check the length of each layer is L
    L = ∑j=n[i-1]..n[i] piece_list[j]
    ∧
    // There is no cut in the danger zones
    ¬∃ m
      with n[i] ≤ m < n[i+1]
      with ∑j=n[i]..m piece_list[j] not in any global_intervals

  ∧

  // Checks not two cuts are too close to each other
  // in consecutive layers of the same beam
  let n_beams = floor(h/n_layers)
  // number of beams that are fully completed
  ∀ k ≤ n_beams, 1 ≤ t < n_layers with t + n_layers * k ≤ h
  // ensure that we have not exceeded number of completed layers
    let i = t + n_layers * k
    ¬∃ m1, m2 with n[i-1] ≤ m1 < n[i]
              with n[i] ≤ m2 < n[i+1]
      let low_sum =∑j=n[i-1]..m1 piece_list[j]
      let up_sum = ∑j=n[i]..m2 piece_list[j]
      abs(low_sum - up_sum) ≤ local_danger
```
