Suggested Solutions to Exercises in "The RAISE Development Method"

Authors:

Chris George,
Anne E. Haxthausen,
Søren Heilmann,
Jan Storbank Pedersen

Preamble

This document is provided free by TERMA Elektronik AS and contains suggested solutions to the exercises in “The RAISE Development Method”. The document may be copied and distributed without restriction. Each solution is identified by the page number of the corresponding exercise in the book.

Possible solutions to exercises in "The RAISE Development Method"

Chapter 2

Page 57

```
scheme
  TYPES =
    class
      type
        Ship,
        Ship_name,
        Berth,
        Occupancy == vacant | occupied_by(occupant : Ship),
        Index = {| i : Int • i ≥ min ∧ max ≥ i |},
        Status_report = Waiting_status* × (Index →m Occupancy_status),
        Waiting_status = Ship_name × Index-set,
        Occupancy_status == vacant | occupied_by(occupant : Ship_name)

      value
        min, max : Int,
        fits : Ship × Berth → Bool,
        indx : Berth → Index,
        name : Ship → Ship_name,
        max_wait : Nat • max_wait > 0

      axiom
        [index_not_empty] max ≥ min,

        [berths_indexable] ∀ b1, b2 : Berth • indx(b1) = indx(b2) ⇒ b1 = b2,

        [ship_names_unique] ∀ s1, s2 : Ship • name(s1) = name(s2) ⇒ s1 = s2
    end
```

```
scheme
  A_HARBOUR1 =
    hide P, B, consistent, waiting_status, possible_berths, berth_status, occupancy_status in
      class
        object
          /* pool of waiting ships */
          P : A_TEST_QUEUE0(T{Ship for Elem, max_wait for bound}),
          /* berths */
          B : A_ARRAY_INIT(T{Occupancy for Elem, vacant for init})

        type Harbour = P.Queue × B.Array

        value
          /* generators */
          arrives : T.Ship × Harbour ~→ Harbour
          arrives(s, (ws, bs)) ≡ (P.enq(s, ws), bs) pre can_arrive(s, (ws, bs)),

          docks : T.Berth × Harbour ~→ Harbour
          docks(b, (ws, bs)) ≡
            let t = (λ s : T.Ship • T.fits(s, b)) in
              let (r, ws′) = P.deq(t, ws) in
                if r = P.fail then
                  (ws′, bs)
                else
                  (ws′, B.change(T.indx(b), T.occupied_by(P.res(r)), bs))
                end
              end
            end
            pre can_dock(b, (ws, bs)),

          leaves : T.Ship × T.Berth × Harbour ~→ Harbour
          leaves(s, b, (ws, bs)) ≡
            (ws, B.change(T.indx(b), T.vacant, bs))
            pre can_leave(s, b, (ws, bs)),

          /* observers */
          waiting : T.Ship × Harbour → Bool
          waiting(s, (ws, bs)) ≡ s ∈ elems P.list_of(ws),

          occupancy : T.Berth × Harbour → T.Occupancy
          occupancy(b, (ws, bs)) ≡ B.apply(T.indx(b), bs),

          status : Harbour → T.Status_report
          status(h) ≡ (waiting_status(h), berth_status(h)),

          /* invariant */
          consistent : Harbour → Bool
          consistent((ws, bs)) ≡
            (
              ∀ s : T.Ship •
                ∼ (waiting(s, (ws, bs)) ∧ is_docked(s, (ws, bs))) ∧
```

```
        (
          ∀ b1, b2 : T.Berth •
            B.apply(T.indx(b1), bs) = T.occupied_by(s) ∧
            B.apply(T.indx(b2), bs) = T.occupied_by(s) ⇒
            b1 = b2
        )
      ),

    is_docked : T.Ship × Harbour → Bool
    is_docked(s, (ws, bs)) ≡
      (∃ b : T.Berth • B.apply(T.indx(b), bs) = T.occupied_by(s)),

    /* guards */
    can_arrive : T.Ship × Harbour → Bool
    can_arrive(s, (ws, bs)) ≡
      ∼ P.is_full(ws) ∧ ∼ waiting(s, (ws, bs)) ∧ ∼ is_docked(s, (ws, bs)),

    can_dock : T.Berth × Harbour → Bool
    can_dock(b, (ws, bs)) ≡
      let t = (λ s : T.Ship • T.fits(s, b)) in
        let r = P.next(t, ws) in r ≠ P.fail ∧ ∼ is_docked(P.res(r), (ws, bs)) end
      end ∧
      B.apply(T.indx(b), bs) = T.vacant,

    can_leave : T.Ship × T.Berth × Harbour → Bool
    can_leave(s, b, (ws, bs)) ≡ B.apply(T.indx(b), bs) = T.occupied_by(s),

    /* auxiliary */
    waiting_status : Harbour → T.Waiting_status*
    waiting_status((ws, bs)) ≡ ⟨ (T.name(s), possible_berths(s)) | s in P.list_of(ws) ⟩,

    possible_berths : T.Ship → T.Index-set
    possible_berths(s) ≡ { T.indx(b) | b : T.Berth • T.fits(s, b) },

    berth_status : Harbour → (T.Index →m T.Occupancy_status)
    berth_status(h) ≡ [ T.indx(b) ↦ occupancy_status(b, h) | b : T.Berth ],

    occupancy_status : T.Berth × Harbour → T.Occupancy_status
    occupancy_status(b, (ws, bs)) ≡
      case B.apply(T.indx(b), bs) of
        T.vacant → T.vacant, T.occupied_by(s) → T.occupied_by(T.name(s))
      end
  end

scheme
  I_HARBOUR1 =
    hide P, B, consistent, waiting_status, possible_berths, berth_status, occupancy_status in
    class
      object
        /* pool of waiting ships */
        P : I_TEST_QUEUE1(T{Ship for Elem, max_wait for bound}),
```

```
  /* berths */
  B : I_ARRAY_INIT(T{Occupancy for Elem, vacant for init})

value
  /* generators */
  arrives : T.Ship ~→ write any Unit
  arrives(s) ≡ P.enq(s) pre can_arrive(s),

  docks : T.Berth ~→ write any Unit
  docks(b) ≡
    let t = (λ s : T.Ship • T.fits(s, b)) in
      let r = P.deq(t) in
        if r = P.fail then skip else B.change(T.indx(b), T.occupied_by(P.res(r))) end
      end
    end
    pre can_dock(b),

  leaves : T.Ship × T.Berth ~→ write any Unit
  leaves(s, b) ≡ B.change(T.indx(b), T.vacant) pre can_leave(s, b),

  /* observers */
  waiting : T.Ship → read any Bool
  waiting(s) ≡ s ∈ elems P.list_of(),

  occupancy : T.Berth → read any T.Occupancy
  occupancy(b) ≡ B.apply(T.indx(b)),

  status : Unit → read any T.Status_report
  status(h) ≡ (waiting_status(h), berth_status(h)),

  /* invariant */
  consistent : Unit → read any Bool
  consistent() ≡
    (
      ∀ s : T.Ship •
        ∼ (waiting(s) ∧ is_docked(s)) ∧
        (
          ∀ b1, b2 : T.Berth •
            B.apply(T.indx(b1)) = T.occupied_by(s) ∧
            B.apply(T.indx(b2)) = T.occupied_by(s) ⇒
            b1 = b2
        )
    ),

  is_docked : T.Ship → read any Bool
  is_docked(s) ≡ (∃ b : T.Berth • B.apply(T.indx(b)) = T.occupied_by(s)),

  /* guards */
  can_arrive : T.Ship → read any Bool
  can_arrive(s) ≡ ∼ P.is_full() ∧ ∼ waiting(s) ∧ ∼ is_docked(s),
```

```
    can_dock : T.Berth → read any Bool
    can_dock(b) ≡
      let t = (λ s : T.Ship • T.fits(s, b)) in
        let r = P.next(t) in r ≠ P.fail ∧ ∼ is_docked(P.res(r)) end
      end ∧
      B.apply(T.indx(b)) = T.vacant,

    can_leave : T.Ship × T.Berth → read any Bool
    can_leave(s, b) ≡ B.apply(T.indx(b)) = T.occupied_by(s),

    /∗ auxiliary ∗/
    waiting_status : Unit → read any T.Waiting_status*
    waiting_status() ≡ ⟨ (T.name(s), possible_berths(s)) | s in P.list_of() ⟩,

    possible_berths : T.Ship → T.Index-set
    possible_berths(s) ≡ { T.indx(b) | b : T.Berth • T.fits(s, b) },

    berth_status : Unit → read any (T.Index →m T.Occupancy_status)
    berth_status() ≡ [ T.indx(b) ↦ occupancy_status(b) | b : T.Berth ],

    occupancy_status : T.Berth → read any T.Occupancy_status
    occupancy_status(b) ≡
      case B.apply(T.indx(b)) of
        T.vacant → T.vacant, T.occupied_by(s) → T.occupied_by(T.name(s))
      end
  end
```

Page 87

We need an index type for the lifts. We could add this to *TYPES* or just define another types module *INDEX*:

```
scheme
  INDEX =
    class
      value
        num_lifts : Nat • num_lifts ≥ 1

      type Lift_index = {| i : Int • i ∈ { 1 .. num_lifts } |}
    end
```

We instantiate this as a global object *I*.

Most of the rest of the development just involves adding an extra parameter of type *I.Lift_index* to the functions. We present the modules in the corresponding order to the single lift example.

```
scheme
  A_LIFTS0 =
    hide movement, door_state, floor, direction, safe in
      class
        type Lifts
```

```
value
  /* generators */
  next : I.Lift_index × T.Requirement × Lifts ~→ Lifts,

  check_buttons : I.Lift_index × Lifts → T.Requirement × Lifts,

  /* observers */
  movement : Lifts → I.Lift_index → T.Movement,

  door_state : Lifts → I.Lift_index × T.Floor → T.Door_state,

  floor : Lifts → I.Lift_index → T.Floor,

  direction : Lifts → I.Lift_index → T.Direction,

  /* derived */
  safe : I.Lift_index × Lifts → Bool
  safe(i, s) ≡
    (
      ∀ f : T.Floor •
        (door_state(s)(i, f) = T.open) = (movement(s)(i) = T.halted ∧ floor(s)(i) = f)
    )

axiom
  [safe_and_useful]
    ∀ s : Lifts, i : I.Lift_index •
      safe(i, s) ⇒
      let (r, s′) = check_buttons(i, s) in
        safe(i, s′) ∧
        let s″ = next(i, r, s′) in
          safe(i, s″) ∧
          (
            movement(s″)(i) = T.halted ⇒
            (T.here(r) ∨ (∼ T.after(r) ∧ ∼ T.before(r))) ∧ floor(s)(i) = floor(s″)(i)
          ) ∧
          (
            movement(s″)(i) = T.moving ⇒
            (T.after(r) ∨ T.before(r)) ∧
            T.is_next_floor(direction(s″)(i), floor(s)(i)) ∧
            floor(s″)(i) = T.next_floor(direction(s″)(i), floor(s)(i))
          ) ∧
          (direction(s)(i) ≠ direction(s″)(i) ⇒ ∼ T.after(r))
        end
      end
end
```

```
scheme
  A_LIFTS1 =
    hide movement, door_state, floor, direction, move, halt, safe in
      class
        type Lifts

        value
          /* generators */
          move : I.Lift_index × T.Direction × T.Movement × Lifts ~→ Lifts,

          halt : I.Lift_index × Lifts → Lifts,

          check_buttons : I.Lift_index × Lifts → T.Requirement × Lifts,

          /* observers */
          movement : Lifts → I.Lift_index → T.Movement,

          door_state : Lifts → I.Lift_index × T.Floor → T.Door_state,

          floor : Lifts → I.Lift_index → T.Floor,

          direction : Lifts → I.Lift_index → T.Direction,

          /* derived */
          next : I.Lift_index × T.Requirement × Lifts ~→ Lifts
          next(i, r, s) ≡
            let d = direction(s)(i) in
              case movement(s)(i) of
                T.halted →
                  case r of
                    T.mk_Requirement(_, true, _) → move(i, d, T.halted, s),
                    T.mk_Requirement(_, _, true) → move(i, T.invert(d), T.halted, s),
                    _ → s
                  end,
                T.moving →
                  case r of
                    T.mk_Requirement(true, _, _) → halt(i, s),
                    T.mk_Requirement(_, false, false) → halt(i, s),
                    T.mk_Requirement(_, true, _) → move(i, d, T.moving, s),
                    T.mk_Requirement(_, _, true) → move(i, T.invert(d), T.moving, s)
                  end
              end
            end
            pre
              (T.after(r) ⇒ T.is_next_floor(direction(s)(i), floor(s)(i))) ∧
              (T.before(r) ⇒ T.is_next_floor(T.invert(direction(s)(i)), floor(s)(i))),

          safe : I.Lift_index × Lifts → Bool
          safe(i, s) ≡
            (
              ∀ f : T.Floor •
```

```
      (door_state(s)(i, f) = T.open) = (movement(s)(i) = T.halted ∧ floor(s)(i) = f)
  )

axiom
  [ movement_move ]
    ∀ s : Lifts, d : T.Direction, m : T.Movement, i, i′ : I.Lift_index •
      movement(move(i, d, m, s))(i′) ≡
        if i = i′ then T.moving else movement(s)(i′) end
        pre T.is_next_floor(d, floor(s)(i)),

  [ door_state_move ]
    ∀ s : Lifts, d : T.Direction, m : T.Movement, f : T.Floor, i, i′ : I.Lift_index •
      door_state(move(i, d, m, s))(i′, f) ≡
        if i = i′ then
          if m = T.halted ∧ floor(s)(i) = f then T.shut else door_state(s)(i, f) end
        else
          door_state(s)(i′, f)
        end
        pre T.is_next_floor(d, floor(s)(i)),

  [ floor_move ]
    ∀ s : Lifts, d : T.Direction, m : T.Movement, i, i′ : I.Lift_index •
      floor(move(i, d, m, s))(i′) ≡
        if i = i′ then T.next_floor(d, floor(s)(i)) else floor(s)(i′) end
        pre T.is_next_floor(d, floor(s)(i)),

  [ direction_move ]
    ∀ s : Lifts, d : T.Direction, m : T.Movement, i, i′ : I.Lift_index •
      direction(move(i, d, m, s))(i′) ≡ if i = i′ then d else direction(s)(i′) end
        pre T.is_next_floor(d, floor(s)(i)),

  [ move_defined ]
    ∀ s : Lifts, d : T.Direction, m : T.Movement, i : I.Lift_index •
      move(i, d, m, s) post true pre T.is_next_floor(d, floor(s)(i)),

  [ movement_halt ]
    ∀ s : Lifts, i, i′ : I.Lift_index •
      movement(halt(i, s))(i′) ≡ if i = i′ then T.halted else movement(s)(i′) end,

  [ door_state_halt ]
    ∀ s : Lifts, f : T.Floor, i, i′ : I.Lift_index •
      door_state(halt(i, s))(i′, f) ≡
        if i = i′ then
          if floor(s)(i) = f then T.open else door_state(s)(i, f) end
        else
          door_state(s)(i′, f)
        end,

  [ floor_halt ] ∀ s : Lifts, i, i′ : I.Lift_index • floor(halt(i, s))(i′) ≡ floor(s)(i′),

  [ direction_halt ]
```

```
      ∀ s : Lifts, i, i′ : I.Lift_index • direction(halt(i, s))(i′) ≡ direction(s)(i′),

    [ check_buttons_ax ]
      ∀ s : Lifts, i : I.Lift_index •
        check_buttons(i, s) as (r, s′)
          post
            movement(s′)(i) = movement(s)(i) ∧
            (∀ f : T.Floor • door_state(s′)(i, f) = door_state(s)(i, f)) ∧
            floor(s′)(i) = floor(s)(i) ∧
            direction(s′)(i) = direction(s)(i) ∧
            (T.after(r) ⇒ T.is_next_floor(direction(s′)(i), floor(s′)(i))) ∧
            (T.before(r) ⇒ T.is_next_floor(T.invert(direction(s′)(i)), floor(s′)(i)))
  end

scheme
  A_DOORS0 =
    class
      type Doors

      value
        /* generators */
        open : I.Lift_index × T.Floor × Doors → Doors,
        close : I.Lift_index × T.Floor × Doors → Doors,
        /* observer */
        door_state : Doors → I.Lift_index × T.Floor → T.Door_state

      axiom
        [ door_state_open ]
          ∀ f, f′ : T.Floor, s : Doors, i, i′ : I.Lift_index •
            door_state(open(i, f, s))(i′, f) ≡
              if i = i′ then
                if f = f′ then T.open else door_state(s)(i, f) end
              else
                door_state(s)(i′, f)
              end,

        [ door_state_close ]
          ∀ f, f′ : T.Floor, s : Doors, i, i′ : I.Lift_index •
            door_state(close(i, f, s))(i′, f) ≡
              if i = i′ then
                if f = f′ then T.shut else door_state(s)(i, f) end
              else
                door_state(s)(i′, f)
              end
    end

scheme
  A_BUTTONS0 =
    class
      type Buttons
```

```
    value
      /∗ generators ∗/
      clear : I.Lift_index × T.Direction × T.Floor × Buttons → Buttons,
      check : I.Lift_index × T.Direction × T.Floor × Buttons → T.Requirement × Buttons

    axiom
      [ check_result ]
        ∀ s : Buttons, d : T.Direction, f : T.Floor, i : I.Lift_index •
          check(i, d, f, s) as (r, s′)
            post
              (T.after(r) ⇒ T.is_next_floor(d, f)) ∧
              (T.before(r) ⇒ T.is_next_floor(T.invert(d), f))
  end

scheme
  A_MOTORS0 =
  class
    type Motors

    value
      /∗ generators ∗/
      move : I.Lift_index × T.Direction × Motors ⥲ Motors,
      halt : I.Lift_index × Motors → Motors,
      /∗ observers ∗/
      direction : Motors → I.Lift_index → T.Direction,
      movement : Motors → I.Lift_index → T.Movement,
      floor : Motors → I.Lift_index → T.Floor

    axiom
      [ direction_move ]
        ∀ s : Motors, d : T.Direction, i, i′ : I.Lift_index •
          direction(move(i, d, s))(i′) ≡ if i = i′ then d else direction(s)(i′) end
            pre T.is_next_floor(d, floor(s)(i)),

      [ movement_move ]
        ∀ s : Motors, d : T.Direction, i, i′ : I.Lift_index •
          movement(move(i, d, s))(i′) ≡ if i = i′ then T.moving else movement(s)(i′) end
            pre T.is_next_floor(d, floor(s)(i)),

      [ floor_move ]
        ∀ s : Motors, d : T.Direction, i, i′ : I.Lift_index •
          floor(move(i, d, s))(i′) ≡
            if i = i′ then T.next_floor(d, floor(s)(i)) else floor(s)(i′) end
            pre T.is_next_floor(d, floor(s)(i)),

      [ move_defined ]
        ∀ s : Motors, d : T.Direction, i : I.Lift_index •
          move(i, d, s) post true pre T.is_next_floor(d, floor(s)(i)),

      [ direction_halt ]
```

```
      ∀ s : Motors, i, i′ : I.Lift_index • direction(halt(i, s))(i′) ≡ direction(s)(i′),

    [ movement_halt ]
      ∀ s : Motors, i, i′ : I.Lift_index •
        movement(halt(i, s))(i′) ≡ if i = i′ then T.halted else movement(s)(i′) end,

    [ floor_halt ] ∀ s : Motors, i, i′ : I.Lift_index • floor(halt(i, s))(i′) ≡ floor(s)(i′)
  end

scheme
  A_LIFTS2_BODY =
    class
      object
        /* motor */
        MS : A_MOTORS0,
        /* doors */
        DS : A_DOORS0,
        /* buttons */
        BS : A_BUTTONS0

      type Lifts = MS.Motors × DS.Doors × BS.Buttons

      value
        /* generators */
        move : I.Lift_index × T.Direction × T.Movement × Lifts ~→ Lifts
        move(i, d, m, (ms, ds, bs)) ≡
          (
            MS.move(i, d, ms),
            if m = T.halted then DS.close(i, MS.floor(ms)(i), ds) else ds end,
            bs
          )
          pre T.is_next_floor(d, MS.floor(ms)(i)),

        halt : I.Lift_index × Lifts → Lifts
        halt(i, (ms, ds, bs)) ≡
          (
            MS.halt(i, ms),
            DS.open(i, MS.floor(ms)(i), ds),
            BS.clear(i, MS.direction(ms)(i), MS.floor(ms)(i), bs)
          ),

        check_buttons : I.Lift_index × Lifts → T.Requirement × Lifts
        check_buttons(i, (ms, ds, bs)) ≡
          let (r, bs′) = BS.check(i, MS.direction(ms)(i), MS.floor(ms)(i), bs) in
            (r, (ms, ds, bs′))
          end,

        /* derived */
        next : I.Lift_index × T.Requirement × Lifts ~→ Lifts
        next(i, r, (ms, ds, bs)) ≡
          let d = MS.direction(ms)(i) in
```

```
          case MS.movement(ms)(i) of
            T.halted →
              case r of
                T.mk_Requirement(_, true, _) → move(i, d, T.halted, (ms, ds, bs)),
                T.mk_Requirement(_, _, true) →
                  move(i, T.invert(d), T.halted, (ms, ds, bs)),
                _ → (ms, ds, bs)
              end,
            T.moving →
              case r of
                T.mk_Requirement(true, _, _) → halt(i, (ms, ds, bs)),
                T.mk_Requirement(_, false, false) → halt(i, (ms, ds, bs)),
                T.mk_Requirement(_, true, _) → move(i, d, T.moving, (ms, ds, bs)),
                T.mk_Requirement(_, _, true) →
                  move(i, T.invert(d), T.moving, (ms, ds, bs))
              end
          end
        end
        pre
          (T.after(r) ⇒ T.is_next_floor(MS.direction(ms)(i), MS.floor(ms)(i))) ∧
          (T.before(r) ⇒ T.is_next_floor(T.invert(MS.direction(ms)(i)), MS.floor(ms)(i))),

      safe : I.Lift_index × Lifts → Bool
      safe(i, (ms, ds, bs)) ≡
        (
          ∀ f : T.Floor •
            (DS.door_state(ds)(i, f) = T.open) =
            (MS.movement(ms)(i) = T.halted ∧ MS.floor(ms)(i) = f)
        )
  end

scheme
  A_LIFTS2 = hide MS, DS, BS, move, halt, safe in A_LIFTS2_BODY

scheme
  A_MOTORS1 =
    class
      type
        Motors = I.Lift_index → Motor,
        Motor :: direction : T.Direction  movement : T.Movement  floor : T.Floor

      value
        /* generators */
        move : I.Lift_index × T.Direction × Motors ~→ I.Lift_index → Motor
        move(i, d, ms)(i′) ≡
          if i = i′ then mk_Motor(d, T.moving, T.next_floor(d, floor(ms(i)))) else ms(i′) end
          pre T.is_next_floor(d, floor(ms(i))),

        halt : I.Lift_index × Motors → I.Lift_index → Motor
        halt(i, ms)(i′) ≡
```

```
        if i = i′ then mk_Motor(direction(ms(i)), T.halted, floor(ms(i))) else ms(i′) end,

      /∗ observers ∗/
      direction : Motors → I.Lift_index → T.Direction
      direction(ms)(i) ≡ direction(ms(i)),

      movement : Motors → I.Lift_index → T.Movement
      movement(ms)(i) ≡ movement(ms(i)),

      floor : Motors → I.Lift_index → T.Floor
      floor(ms)(i) ≡ floor(ms(i))
  end
```

```
scheme
  A_DOORS1 =
    class
      type Doors = I.Lift_index × T.Floor → T.Door_state

      value
        /∗ generators ∗/
        open : I.Lift_index × T.Floor × Doors → I.Lift_index × T.Floor → T.Door_state
        open(i, f, s)(i′, f′) ≡
          if i = i′ then if f = f′ then T.open else s(i, f′) end else s(i′, f′) end,

        close : I.Lift_index × T.Floor × Doors → I.Lift_index × T.Floor → T.Door_state
        close(i, f, s)(i′, f′) ≡
          if i = i′ then if f = f′ then T.shut else s(i, f′) end else s(i′, f′) end,

        /∗ observer ∗/
        door_state : Doors → I.Lift_index × T.Floor → T.Door_state
        door_state(s) ≡ s
    end
```

In *A_BUTTONS1* we make the change that a floor button is only cleared if the lift is traveling in the appropriate direction.

```
scheme
  A_BUTTONS1 =
    hide required_here, required_beyond in
      class
        type
          Buttons =
            (I.Lift_index × T.Floor → T.Button_state) ×
              (T.Lower_floor → T.Button_state) × (T.Upper_floor → T.Button_state)

        value
          /∗ generators ∗/
          clear : I.Lift_index × T.Direction × T.Floor × Buttons → Buttons
          clear(i, d, f, (lift, up, down)) ≡
            (
```

```
        λ (i′, f′) : I.Lift_index × T.Floor •
          if i = i′ then if f = f′ then T.clear else lift(i, f′) end else lift(i′, f′) end,
        λ f′ : T.Lower_floor •
          if f = f′ ∧ d = T.up then T.clear else up(f′) end,
        λ f′ : T.Upper_floor •
          if f = f′ ∧ d = T.down then T.clear else down(f′) end
      ),

    check :
      I.Lift_index × T.Direction × T.Floor × Buttons → T.Requirement × Buttons,

    /* observers */
    required_here : I.Lift_index × T.Direction × T.Floor × Buttons → Bool
    required_here(i, d, f, (lift, up, down)) ≡
      lift(i, f) = T.lit ∨
      d = T.up ∧
      (
        f < T.max_floor ∧ up(f) = T.lit ∨
        f > T.min_floor ∧ down(f) = T.lit ∧ ∼ required_beyond(i, d, f, (lift, up, down))
      ) ∨
      d = T.down ∧
      (
        f > T.min_floor ∧ down(f) = T.lit ∨
        f < T.max_floor ∧ up(f) = T.lit ∧ ∼ required_beyond(i, d, f, (lift, up, down))
      ),

    required_beyond : I.Lift_index × T.Direction × T.Floor × Buttons → Bool
    required_beyond(i, d, f, s) ≡
      T.is_next_floor(d, f) ∧
      let f′ = T.next_floor(d, f) in
        required_here(i, d, f′, s) ∨ required_beyond(i, d, f′, s)
      end

  axiom
    [check_result]
      ∀ s : Buttons, d : T.Direction, f : T.Floor, i : I.Lift_index •
        check(i, d, f, s) as (r, s′)
          post
            r =
            T.mk_Requirement
              (
                required_here(i, d, f, s),
                required_beyond(i, d, f, s),
                required_beyond(i, T.invert(d), f, s)
              )
end

scheme
  C_LIFTS2 =
    hide MS, DS, BS, move, halt in
      class
```

```
object
  /* motor */
  MS : C_MOTORS1,
  /* doors */
  DS : C_DOORS1,
  /* buttons */
  BS : C_BUTTONS1

value
  /* methods */
  move : I.Lift_index × T.Direction × T.Movement → in any out any Unit
  move(i, d, m) ≡
    if m = T.halted then
      let f = MS.floor(i) in BS.clear(i, d, f) ; DS.close(i, f) end
    end ;
    MS.move(i, d),

  halt : I.Lift_index → in any out any Unit
  halt(i) ≡
    let f = MS.floor(i), d = MS.direction(i) in
      BS.clear(i, d, f) ; MS.halt(i) ; DS.open(i, f)
    end,

  check_buttons : I.Lift_index → in any out any T.Requirement
  check_buttons(i) ≡ BS.check(i, MS.direction(i), MS.floor(i)),

  next : I.Lift_index × T.Requirement → in any out any Unit
  next(i, r) ≡
    let d = MS.direction(i) in
      case MS.movement(i) of
        T.halted →
          case r of
            T.mk_Requirement(_, true, _) → move(i, d, T.halted),
            T.mk_Requirement(_, _, true) → move(i, T.invert(d), T.halted),
            _ → skip
          end,
        T.moving →
          case r of
            T.mk_Requirement(true, _, _) → halt(i),
            T.mk_Requirement(_, false, false) → halt(i),
            T.mk_Requirement(_, true, _) → move(i, d, T.moving),
            T.mk_Requirement(_, _, true) → move(i, T.invert(d), T.moving)
          end
      end
    end,

  /* initial */
  init : Unit → in any out any write any Unit
  init() ≡ MS.init() ∥ DS.init() ∥ BS.init(),

  /* control */
```

```
            lift : Unit → in any out any Unit
            lift() ≡ ∥ { while true do next(i, check_buttons(i)) end | i : I.Lift_index }
        end

scheme
  C_MOTORS1 =
    hide MS in
      class
        object MS[ i : I.Lift_index ] : C_MOTOR1

        value
          /* initial */
          init : Unit → in any out any write any Unit
          init() ≡ ∥ { MS[ i ].init() | i : I.Lift_index },

          /* methods */
          move : I.Lift_index × T.Direction → in any out any Unit
          move(i, d) ≡ MS[ i ].move(d),

          halt : I.Lift_index → in any out any Unit
          halt(i) ≡ MS[ i ].halt(),

          direction : I.Lift_index → in any out any T.Direction
          direction(i) ≡ MS[ i ].direction(),

          floor : I.Lift_index → in any out any T.Floor
          floor(i) ≡ MS[ i ].floor(),

          movement : I.Lift_index → in any out any T.Movement
          movement(i) ≡ MS[ i ].movement()
      end

scheme
  C_DOORS1 =
    hide DS in
      class
        object DS[ i : I.Lift_index, f : T.Floor ] : C_DOOR1

        value
          /* initial */
          init : Unit → in any out any write any Unit
          init() ≡ ∥ { DS[ i, f ].init() | i : I.Lift_index, f : T.Floor },

          /* methods */
          open : I.Lift_index × T.Floor → in any out any Unit
          open(i, f) ≡ DS[ i, f ].open(),

          close : I.Lift_index × T.Floor → in any out any Unit
          close(i, f) ≡ DS[ i, f ].close(),
```

```
        door_state : I.Lift_index × T.Floor → in any out any T.Door_state
        door_state(i, f) ≡ DS[i, f].door_state()
    end

scheme
  C_BUTTONS1 =
    hide LB, UB, DB, required_here, required_beyond in
      class
        object
          /* lift buttons */
          LB[i : I.Lift_index, f : T.Floor] : C_BUTTON1,
          /* up buttons */
          UB[f : T.Lower_floor] : C_BUTTON1,
          /* down buttons */
          DB[f : T.Upper_floor] : C_BUTTON1

        value
          /* initial */
          init : Unit → in any out any write any Unit
          init() ≡
            || { LB[i, f].init() | i : I.Lift_index, f : T.Floor }
            ||
            || { UB[f].init() | f : T.Lower_floor }
            ||
            || { DB[f].init() | f : T.Upper_floor },

          /* methods */
          clear : I.Lift_index × T.Direction × T.Floor → in any out any Unit
          clear(i, d, f) ≡
            LB[i, f].clear() ;
            if f < T.max_floor ∧ d = T.up then UB[f].clear() end ;
            if f > T.min_floor ∧ d = T.down then DB[f].clear() end,

          check : I.Lift_index × T.Direction × T.Floor → in any out any T.Requirement
          check(i, d, f) ≡
            T.mk_Requirement
              (
                required_here(i, d, f), required_beyond(i, d, f), required_beyond(i, T.invert(d), f)
              ),

          required_here : I.Lift_index × T.Direction × T.Floor → in any out any Bool
          required_here(i, d, f) ≡
            LB[i, f].check() = T.lit ∨
            d = T.up ∧
            (
              f < T.max_floor ∧ UB[f].check() = T.lit ∨
              f > T.min_floor ∧ DB[f].check() = T.lit ∧ ∼ required_beyond(i, d, f)
            ) ∨
            d = T.down ∧
            (
              f > T.min_floor ∧ DB[f].check() = T.lit ∨
```

$$
\begin{array}{l}
\qquad\qquad\quad \mathrm{f} < \mathrm{T.max\_floor} \wedge \mathrm{UB[f].check()} = \mathrm{T.lit} \wedge \sim \mathrm{required\_beyond(i,\ d,\ f)}\\
\qquad\qquad \mathrm{),}\\
\\
\qquad\qquad \mathrm{required\_beyond : I.Lift\_index} \times \mathrm{T.Direction} \times \mathrm{T.Floor} \rightarrow \textbf{in any out any Bool}\\
\qquad\qquad \mathrm{required\_beyond(i,\ d,\ f)} \equiv\\
\qquad\qquad\quad \mathrm{T.is\_next\_floor(d,\ f)} \wedge\\
\qquad\qquad\quad \textbf{let}\ \mathrm{f}' = \mathrm{T.next\_floor(d,\ f)}\ \textbf{in}\ \mathrm{required\_here(i,\ d,\ f')} \vee \mathrm{required\_beyond(i,\ d,\ f')}\ \textbf{end}\\
\qquad \textbf{end}
\end{array}
$$

C_MOTOR1, *C_DOOR1* and *C_BUTTON1* are unchanged from the original single lift specification.

Page 95

For observational axioms characterizing an unbounded set, see the standard specification *A_SET* in appendix A.2.

A bounded set cannot be characterized by observational axioms with these generators and observers. An additional observer, such as *count*, is needed.

For stacks and queues, even when unbounded, it is not possible to define them with observational axioms unless observers additional to the usual ones are added.

In particular, the commonly used axiom for stacks

$$\forall\ \mathrm{e : E.Elem,\ st : Stack} \bullet \mathrm{pop(push(e,\ st))} \equiv \mathrm{(e,\ st)}$$

is not observational because it relates two generators.

See the text following the exercise for further discussion.

Page 142

We suggested making *deq* and *next* total since, in a concurrent version, their "test" parameters need to be passed to the main process before success or failure can be determined.

We present a development based on RSL lists.

$$
\begin{array}{l}
\textbf{scheme}\\
\quad \mathrm{A\_TEST\_QUEUE0(P : ELEM\_BOUND)} =\\
\qquad \textbf{class}\\
\qquad\quad \textbf{type}\\
\qquad\qquad \mathrm{Queue,}\\
\qquad\qquad \mathrm{List\_of\_Queue} = \{|\ \mathrm{l : P.Elem^{*}} \bullet \textbf{len}\ \mathrm{l} \leq \mathrm{P.bound}\ |\},\\
\qquad\qquad \mathrm{Deq\_result} == \mathrm{fail} \mid \mathrm{ok(res : P.Elem)}\\
\\
\qquad\quad \textbf{value}\\
\qquad\qquad /*\ \mathrm{generators}\ */\\
\qquad\qquad \mathrm{empty : Queue,}\\
\\
\qquad\qquad \mathrm{enq : P.Elem} \times \mathrm{Queue} \stackrel{\sim}{\rightarrow} \mathrm{Queue,}\\
\\
\qquad\qquad \mathrm{deq : (P.Elem} \rightarrow \textbf{Bool}) \times \mathrm{Queue} \rightarrow \mathrm{Deq\_result} \times \mathrm{Queue,}\\
\\
\qquad\qquad /*\ \mathrm{observer}\ */\\
\qquad\qquad \mathrm{list\_of : Queue} \rightarrow \mathrm{List\_of\_Queue,}
\end{array}
$$

```
      /∗ derived ∗/
      next : (P.Elem → Bool) × Queue → Deq_result
      next(t, q) ≡ let (r, q′) = deq(t, q) in r end,

      is_full : Queue → Bool
      is_full(q) ≡ len list_of(q) = P.bound

    axiom
      [ list_of_empty ] list_of(empty) ≡ ⟨⟩,

      [ list_of_enq ]
        ∀ e : P.Elem, q : Queue • list_of(enq(e, q)) ≡ list_of(q) ⌢ ⟨e⟩ pre ∼ is_full(q),

      [ deq_ax ]
        ∀ q : Queue, t : P.Elem → Bool •
          deq(t, q) as (r, q′)
            post
              (
                ∃ el, el′ : List_of_Queue, e′ : P.Elem •
                  list_of(q) = el ⌢ ⟨e′⟩ ⌢ el′ ∧
                  (∀ e : P.Elem • e ∈ elems el ⇒ ∼ t(e)) ∧
                  t(e′) ∧ r = ok(e′) ∧ list_of(q′) = el ⌢ el′
              ) ∨
              (
                (∀ e : P.Elem • e ∈ elems list_of(q) ⇒ ∼ t(e)) ∧
                r = fail ∧ list_of(q′) = list_of(q)
              ),

      [ enq_defined ] ∀ e : P.Elem, q : Queue • enq(e, q) post true pre ∼ is_full(q)
  end

scheme
  A_TEST_QUEUE1(P : ELEM_BOUND) =
    class
      type
        Queue = List_of_Queue,
        List_of_Queue = {| l : P.Elem∗ • len l ≤ P.bound |},
        Deq_result == fail | ok(res : P.Elem)

      value
        /∗ generators ∗/
        empty : Queue = ⟨⟩,

        enq : P.Elem × Queue →~ Queue
        enq(e, q) ≡ q ⌢ ⟨e⟩ pre ∼ is_full(q),

        deq : (P.Elem → Bool) × Queue → Deq_result × Queue
        deq(t, q) ≡
          case q of
            ⟨⟩ → (fail, q),
            ⟨h⟩ ⌢ el →
```

```
        if t(h) then (ok(h), el) else let (r, q′) = deq(t, el) in (r, ⟨h⟩ ⌢ q′) end end
      end,

    /∗ observers ∗/
    next : (P.Elem → Bool) × Queue → Deq_result
    next(t, q) ≡
      case q of ⟨⟩ → fail, ⟨h⟩ ⌢ el → if t(h) then ok(h) else next(t, el) end end,

    list_of : Queue → List_of_Queue
    list_of(q) ≡ q,

    is_full : Queue → Bool
    is_full(q) ≡ len q = P.bound
  end

scheme
  I_TEST_QUEUE1(P : ELEM_BOUND) =
    hide A, queue in
      class
        object A : A_TEST_QUEUE1(P)

        type List_of_Queue = A.List_of_Queue,  Deq_result = A.Deq_result

        variable queue : List_of_Queue := ⟨⟩

        value
          /∗ generators ∗/
          empty : Unit → write queue Unit
          empty() ≡ queue := ⟨⟩,

          enq : P.Elem ~→ write queue Unit
          enq(e) ≡ queue := queue ⌢ ⟨e⟩ pre ∼ is_full(),

          deq : (P.Elem → Bool) → write queue Deq_result
          deq(t) ≡ let (r, q′) = A.deq(t, queue) in queue := q′ ; r end,

          /∗ observers ∗/
          next : (P.Elem → Bool) → read queue Deq_result
          next(t) ≡ A.next(t, queue),

          list_of : Unit → read queue List_of_Queue
          list_of() ≡ queue,

          is_full : Unit → read queue Bool
          is_full() ≡ len queue = P.bound,

          /∗ auxiliary ∗/
          fail : Deq_result = A.fail,

          ok : P.Elem → Deq_result = A.ok,
```

```
        res : Deq_result ~→ P.Elem = A.res

      axiom [initial] initialise post queue = ⟨⟩
    end

scheme
  C_TEST_QUEUE1(P : ELEM_BOUND) =
    hide I, CH, main in
      class
        object I : I_TEST_QUEUE1(P)

        type List_of_Queue = I.List_of_Queue,  Deq_result = I.Deq_result

        object
          CH :
            class
              channel
                enq : P.Elem,
                deq, next : P.Elem → Bool,
                deq_res, next_res : Deq_result,
                list_of : List_of_Queue,
                is_full : Bool
            end

        value
          /* main */
          main : Unit → in any out any write any Unit
          main() ≡
            while true do
              if ∼ I.is_full() then let e = CH.enq? in I.enq(e) end else stop end
              ⌷
              let t = CH.deq? in CH.deq_res ! I.deq(t) end
              ⌷
              let t = CH.next? in CH.next_res ! I.next(t) end
              ⌷
              CH.list_of ! I.list_of()
              ⌷
              CH.is_full ! I.is_full()
            end,

          /* initial */
          empty : Unit → in any out any write any Unit
          empty() ≡ I.empty() ; main(),

          /* generators */
          enq : P.Elem → in any out any Unit
          enq(e) ≡ CH.enq ! e,

          deq : (P.Elem → Bool) → in any out any Deq_result
          deq(t) ≡ CH.deq ! t ; CH.deq_res?,
```

```
    /* observers */
    next : (P.Elem → Bool) → in any out any Deq_result
    next(t) ≡ CH.next ! t ; CH.next_res?,

    list_of : Unit → in any out any List_of_Queue
    list_of() ≡ CH.list_of?,

    is_full : Unit → in any out any Bool
    is_full() ≡ CH.is_full?,

    /* auxiliary */
    fail : Deq_result = I.fail,

    ok : P.Elem → Deq_result = I.ok,

    res : Deq_result ~→ P.Elem = I.res
  end
```

Page 154

```
 ⌞(∀ e : P.Elem •
     let l = list_of() in enq(e) ; list_of() ≡ enq(e) ; l ⌢ ⟨e⟩ pre ∼ is_full() end) ≡
   (∀ e : P.Elem •
     enq(e) ; list_of() ≡ let l = list_of() in enq(e) ; l ⌢ ⟨e⟩ end pre ∼ is_full())⌟
is_pre, simplify :
 ⌞(∀ e : P.Elem •
     let l = list_of() in ∼ is_full() ⇒ (enq(e) ; list_of() ≡ enq(e) ; l ⌢ ⟨e⟩) end) ≡
   (∀ e : P.Elem •
     enq(e) ; list_of() ≡ let l = list_of() in enq(e) ; l ⌢ ⟨e⟩ end pre ∼ is_full())⌟
implies_expansion :
 ⌞(∀ e : P.Elem •
     let l = list_of() in if ∼ is_full() then enq(e) ; list_of() ≡ enq(e) ; l ⌢ ⟨e⟩ else true end end) ≡
   (∀ e : P.Elem •
     enq(e) ; list_of() ≡ let l = list_of() in enq(e) ; l ⌢ ⟨e⟩ end pre ∼ is_full())⌟
let_if2 :
 ⌞(∀ e : P.Elem •
     if let l = list_of() in ∼ is_full() end
     then let l = list_of() in enq(e) ; list_of() ≡ enq(e) ; l ⌢ ⟨e⟩ end
     else let l = list_of() in true end end) ≡
   (∀ e : P.Elem •
     enq(e) ; list_of() ≡ let l = list_of() in enq(e) ; l ⌢ ⟨e⟩ end pre ∼ is_full())⌟
let_is :
 ⌞(∀ e : P.Elem •
     if let l = list_of() in ∼ is_full() end
     then let l = list_of() in enq(e) ; list_of() end ≡ let l = list_of() in enq(e) ; l ⌢ ⟨e⟩ end
     else let l = list_of() in true end end) ≡
   (∀ e : P.Elem •
     enq(e) ; list_of() ≡ let l = list_of() in enq(e) ; l ⌢ ⟨e⟩ end pre ∼ is_full())⌟
let_absorption1, let_absorption1, let_absorption1 :
 ⌞(∀ e : P.Elem •
```

$\quad\quad \textbf{if} \sim \text{is\_full}() \textbf{ then } \text{enq(e) ; list\_of()} \equiv \textbf{let } \text{l = list\_of()} \textbf{ in } \text{enq(e) ; l} \frown \langle\text{e}\rangle \textbf{ end else true end}) \equiv$

$\quad (\forall\ \text{e : P.Elem} \bullet \text{enq(e) ; list\_of()} \equiv \textbf{let } \text{l = list\_of()} \textbf{ in } \text{enq(e) ; l} \frown \langle\text{e}\rangle \textbf{ end pre} \sim \text{is\_full()})\lrcorner$

implies_expansion :

$\llcorner(\forall\ \text{e : P.Elem} \bullet$

$\quad\quad \sim \text{is\_full()} \Rightarrow (\text{enq(e) ; list\_of()} \equiv \textbf{let } \text{l = list\_of()} \textbf{ in } \text{enq(e) ; l} \frown \langle\text{e}\rangle \textbf{ end})) \equiv$

$\quad (\forall\ \text{e : P.Elem} \bullet$

$\quad\quad \text{enq(e) ; list\_of()} \equiv \textbf{let } \text{l = list\_of()} \textbf{ in } \text{enq(e) ; l} \frown \langle\text{e}\rangle \textbf{ end pre} \sim \text{is\_full()})\lrcorner$

is_pre, simplify, **qed**

Chapter 3

Page 168

Yes, (2) does implement (1). Contradictory modules (with an implementing signature) are always implementations. Their theory is **false**, and from this one can prove any property of a module being implemented.

Page 169

1. Abstract version:

   value depth : Queue → **Nat**
   axiom
   [depth_empty] depth(empty) = 0,
   [depth_enq]
   ∀ el : El, q : Queue • depth(enq(el, q)) ≡ depth(q) + 1

   Concrete version:

   value
   depth : Queue → **Nat**
   depth(q) ≡ **len** q

2. We have to show in the concrete version the truth of

   (a) *empty_enq*
   (b) *Queue_induction*
   (c) *depth_empty*
   (d) *depth_enq*

   The first, third and fourth all follow immediately by unfolding and using properties of RSL lists. For instance, for *depth_empty* we need to prove

   depth(empty) ≡ 0
   i.e.
   len ⟨⟩ ≡ 0

   and this is a property of the RSL **len** function.

   For *Queue_induction* we need to prove

   ∀ p : Queue → **Bool** •
   (p(empty) ∧ (∀ e : El, q : Queue • p(q) ⇒ p(enq(e, q)))) ⇒
   (∀ q : Queue • p(q))
   i.e.

$$\forall\ p : El^* \to \mathbf{Bool} \bullet$$
$$(p(\langle\rangle) \wedge (\forall\ e : El,\ q : El^* \bullet p(q) \Rightarrow p(q^\frown\langle e\rangle))) \Rightarrow$$
$$(\forall\ q : El^* \bullet p(q))$$

and this is precisely the *all_list_right_induction* property we could assume.

Page 170

(1) expands into

type Colour
value red, green : Colour
axiom [red_green] red ≠ green

(2) expands into

type Colour
value red, green, blue : Colour
axiom
  [red_green] red ≠ green,
  [red_blue] red ≠ blue,
  [green_blue] green ≠ blue,
  [Colour_induction]
    $\forall\ p : Colour \to \mathbf{Bool} \bullet$
      $p(red) \wedge p(green) \wedge p(blue) \Rightarrow (\forall\ c : Colour \bullet p(c))$

Clearly (2) implements (1); its signature includes that of (1) and it has the only property of (1).

This would not be the case if the wildcard were removed from (1). (1) would then have an induction axiom

  [Colour_induction]
    $\forall\ p : Colour \to \mathbf{Bool} \bullet$
      $p(red) \wedge p(green) \Rightarrow (\forall\ c : Colour \bullet p(c))$ (3)

and (3) does not hold in (2). For example, from (3) one can prove that red and green are the only colours, by taking the predicate p to be

$$\lambda\ c : Colour \bullet c = red \vee c = green$$

and this is clearly not the case for (2).

Page 172

Taking the hint that (2) defines a type like "list", and that (1) could be like "set", consider the property

$$\forall\ e : Elem \bullet insert(e, insert(e, empty)) \neq insert(e, empty) \qquad (3)$$

(3) is true for (2) because, if it were false, so that the inequality in (3) were an equality, we could apply *rest* to each side of the equality and deduce

$$\forall\ e : Elem \bullet insert(e, empty) = empty$$

which is the negation of the disjointness axiom for *Collection*. (3) is not necessarily true for (1). In fact it is possible to implement (1), by defining *Collection* as *Elem*-**set**, so that (3) is false. So (3) is a suitable property to show that (2) does not conservatively extend (1).

Page 173

All of the developments are implementations except the second. The second does not preserve the property $x \geq 0$.

Page 178

value
  f : **Int** → **Int**
  f(x) ≡ **abs** x,

  g : **Int*** $\xrightarrow{\sim}$ **Int**
  g(x) ≡
    case x **of**
      ⟨h⟩ → h,
      ⟨h⟩^t → **let** max = g(t) **in if** max ≥ h **then** max **else** h **end end**
    end
  pre x ≠ ⟨⟩

Page 178

We need to show that the axiom

$$\forall\ \mathrm{x} : \mathbf{Nat} \bullet \mathrm{factorial(x)} \equiv \mathbf{if}\ \mathrm{x} = 0\ \mathbf{then}\ 1\ \mathbf{else}\ \mathrm{x} * \mathrm{factorial(x{-}1)}\ \mathbf{end} \qquad (1)$$

is true for the second definition of *factorial*. We do this by induction on the **Nat** x.

- Base case:

  factorial(0) ≡ **if** 0 = 0 **then** 1 **else** ... **end**
  i.e. (unfolding on the left)
  if 0 < 2 **then** 1 **else** ... **end** ≡ **if** 0 = 0 **then** 1 **else** ... **end**

  which is clearly true.

- Inductive case:

  The inductive hypothesis is (1). We need to show

  factorial(x+1) ≡ **if** x+1 = 0 **then** 1 **else** (x+1) ∗ factorial((x+1)−1) **end**
  i.e.
  factorial(x+1) ≡ (x+1) ∗ factorial(x)
  i.e. (unfolding on the left)
  if x + 1 < 2 **then** 1 **else** (x+1) ∗ ((x+1)−1) ∗ factorial((x+1)−2) **end** ≡
    (x+1) ∗ factorial(x)
  i.e.
  if x = 0 **then** 1 **else** (x+1) ∗ x ∗ factorial(x−1) **end** ≡ (x+1) ∗ factorial(x)
  i.e.
  (x+1) ∗ (**if** x = 0 **then** 1 **else** x ∗ factorial(x−1) **end**) ≡ (x+1) ∗ factorial(x)
  i.e. (by inductive hypothesis)
  (x+1) ∗ (factorial(x)) ≡ (x+1) ∗ factorial(x)

  which is immediate.

Page 181

```
value
  sum : Int* → Int
  sum(x) ≡
    local
      variable to_do : Int* := x, so_far : Int := 0
    in
      while to_do ≠ ⟨⟩ do
        so_far := so_far + hd to_do ;
        to_do := tl to_do
      end ;
      so_far
    end
```

Page 183

```
value
  test1, test2, test3 : Event → Bool,
  filter : Event* → Event*
  filter(el) ≡
    local
      value
        filt : (Event → Bool) → Event* → Event*
        filt(test)(el) ≡
          case el of
            ⟨⟩ → ⟨⟩,
            ⟨h⟩⌢t → if test(h) then ⟨h⟩⌢filter(t) else filter(t) end
          end
    in
      filt(test3)(filt(test2)(filt(test1)(el)))
    end
```

It is also possible to avoid the use of functions taking functions as arguments by defining *filt1*, *filt2* and *filt3* functions and composing *filter* as

```
  filt3(filt2(filt1(el)))
```

If separate hardware is available for each test then these hardware units can presumably run in parallel, so a concurrent decomposition would be more appropriate.

Page 185

```
variable wl : Word* := ⟨⟩
value
  words : Text → write wl Unit
  words(t) ≡
    local variable to_do : Text := t,  w : Text := ⟨⟩
    in
      while to_do ≠ ⟨⟩ do
        let c = hd to_do in
          if is_word_char(c) then w := w ⌢ ⟨c⟩
          else
            if w ≠ ⟨⟩ then wl := wl ⌢ ⟨w⟩ ; w := ⟨⟩ end
```

```
        end
      end ;
      to_do := tl to_do
    end ;
    if w ≠ ⟨⟩ then wl := wl ^ ⟨w⟩ end
  end
```

Note that we cannot use *Word* as the type of *w*, since *w* can contain an empty list.

Page 185

```
value
  check : Text → Word*
  check(t) ≡
    local
      variable wl : Word* := ⟨⟩
      channel next : Word, eot : Unit
      value words : Text → out next, eot Unit
      axiom
        words("") ≡ eot!(),
        ∀ t : Text •
          words(t) ≡ words(tl t) pre t ≠ "" ∧ ∼is_word_char(hd t),
        ∀ w : Word, r, t : Text •
          words(t) ≡ next!w ; words(r)
          pre t = w^r ∧ is_sub_word(w, t)
      value
        check_words : Unit → write wl in next, eot Unit
          check_words() ≡
            eot?
            []
            let w = next? in
              if ∼is_in_dict(w) then wl := wl ^ ⟨w⟩ end
            end ; check_words()
    in (words(t) ‖ check_words()) ; wl end
```

Page 189

```
value
  words : Text → Word*
  words(t) ≡
    case t of
      ⟨⟩ → ⟨⟩,
      ⟨h⟩^t′ →
        if is_word_char(h) then
          let (w,t″) = complete(⟨h⟩,t′) in ⟨w⟩^words(t″) end
        else words(t′) end
    end,
  complete : Word × Text → Word × Text
  complete(w,t) ≡
    case t of
      ⟨⟩ → (w,t),
      ⟨h⟩^t′ → if is_word_char(h) then complete(w^⟨h⟩,t′) else (w,t′) end
```

end

Page 192

Define for convenience a new, additional meaning of $\leq$:

value
$\leq$: **Int** $\times$ **Int*** $\rightarrow$ **Bool**
$\mathrm{i} \leq \mathrm{el} \equiv (\forall\ \mathrm{j} : \textbf{Int} \bullet \mathrm{j} \in \textbf{elems}\ \mathrm{el} \Rightarrow \mathrm{i} \leq \mathrm{j})$

It is easy to prove some useful properties of this operator:

$\mathrm{i} \leq \langle\rangle$
$\mathrm{i} \leq \mathrm{j} \wedge \mathrm{j} \leq \mathrm{el} \Rightarrow \mathrm{i} \leq \mathrm{el}$
$\mathrm{i} \leq \mathrm{el} \wedge \mathrm{i} \leq \mathrm{el}' \equiv \mathrm{i} \leq \mathrm{el}^\frown\mathrm{el}'$

It is then straightforward to show that

$$\begin{aligned} &\mathrm{left}\ {}^\frown\ \mathrm{right}\ {}^\frown\ \mathrm{to\_do} = \mathrm{l\_hook} \wedge \\ &\mathrm{right} \neq \langle\rangle \wedge \\ &\textbf{hd}\ \mathrm{right} \leq \mathrm{left}\ {}^\frown\ \textbf{tl}\ \mathrm{right} \end{aligned} \qquad (1)$$

is invariant for the loop in *split*. An expression *eb* is invariant for a convergent expression *eu* if

$\Box$ eu **post** eb **pre** eb

(1) is also easily established for the initial call, and on termination (with *to_do* empty) satisfies the postcondition for split, taking *l2* and *l3* to be *left* and **tl** *right* respectively.

The loop is terminating since the finite sequence *to_do* is reduced in length by one on each iteration and the loop then terminates. The body of the loop is convergent since the only partial operators are **hd** and **tl** applied to *right* (invariantly non-empty) and **tl** applied to *to_do* (non-empty within the loop).

Page 194

We show the justification of *enq_ax*. Ignoring quantification, we have to show, when *I.is_full()* is false:

main() ⫲ enq(e) ≡ I.enq(e) ; main()

We start with the left-hand side:

main() ⫲ enq(e)
≡
(**if** ∼ I.is_full() **then let** e = CH.enq? **in** I.enq(e) **end else stop end**
⌷ ...) ; main() ⫲ CH.enq ! e
≡
let e = e **in** I.enq(e) **end** ; main()
≡
I.enq(e) ; main()

as required.

Page 196

scheme I_OPB(E : ELEM) =
  hide Opb, empty, opb, val, buffer **in**
  class
    type Opb == empty | opb(val : E.Elem)
    variable buffer : Opb := empty

```
  value
    /* generators */
    put : E.Elem ~→ write any Unit
    put(e) ≡ buffer := opb(e) pre is_empty(),
    get : Unit ~→ write any E.Elem
    get() ≡ let e = val(buffer) in buffer := empty ; e end pre ∼is_empty(),
    /* observer */
    is_empty : Unit → read any Bool
    is_empty() ≡ buffer = empty
end
```

Page 196

```
value
  can_be_true : Int × Int × (Int → Bool) → Bool
  can_be_true(i, j, p) ≡
    local
      variable result : Bool := false
    in
      for k in ⟨i .. j⟩ do
        result := result ∨ p(k)
      end ;
      result
    end
```

Specification of *always_true*:

```
value
  always_true : Int × Int × (Int → Bool) → Bool
  always_true(i, j, p) ≡ (∀ k : Int • i ≤ k ∧ k ≤ j ⇒ p(k))
```

Implementation:

```
value
  always_true : Int × Int × (Int → Bool) → Bool
  always_true(i, j, p) ≡
    local
      variable result : Bool := true
    in
      for k in ⟨i .. j⟩ do
        result := result ∧ p(k)
      end ;
      result
    end
```

A version of *can_be_true* using two variables and a while loop:

```
value
  can_be_true : Int × Int × (Int → Bool) → Bool
  can_be_true(i, j, p) ≡
    local
      variable found : Bool := false,  k : Int := i
    in
      while ∼found ∧ k ≤ j do
        found := p(k) ; k := k + 1
```

```
    end ;
    found
  end
```

To decide which version is "preferable" there are at least three issues to consider:

- clarity
- correctness
- efficiency of translated code

This is presumably not an initial specification, since both versions are developments of an implicit specification, and so clarity is less important. It should be apparent that the implicit specification is the clearest.

Correctness is important; we have to justify, more or less formally, that an explicit version is correct, by showing it implements the implicit version. The for loop is perhaps more natural an implementation and more obviously correct, and so preferable if we do not prove correctness. For a formal proof of correctness the two forms are probably similar.

Use of a while loop in *can_be_true* allows the loop to be exited as soon as a true value of *p(k)* is found, at the cost of slightly more computation in each iteration. Whether this is more efficient depends on the probability of finding a true value of *p(k)*.

Page 200

See the version of *split* on page 191 of the book.

Page 203

```
scheme C_OPB(E : ELEM) =
  hide I, CH, main in
  class
    object
      I : I_OPB(E),
      CH : class channel put, get : E.Elem,  is_empty : Bool end
    value
      /* main */
      main : Unit → in any out any write any Unit
      main() ≡
        while true do
          if I.is_empty() then let e = CH.put? in I.put(e) end else stop end
          ⌷
          if ~I.is_empty then CH.get!I.get() else stop end
          ⌷
          CH.is_empty!I.is_empty()
        end,
      /* initial */
      init : Unit → in any out any write any Unit
      init() ≡ I.initialise ; main(),
      /* generators */
      put : E.Elem → in any out any Unit
      put(e) ≡ CH.put!e,
      get : Unit → in any out any E.Elem
      get() ≡ CH.get?,
```

```
    /* observer */
    is_empty : Unit → in any out any Bool
    is_empty() ≡ CH.is_empty?
  end
```

Page 204

```
value
  test1, test2, test3 : Event → Bool,
  filter : Event* → Event*
  filter(el) ≡
    local
      type
        Chan_index = {| i : Int • i ∈ {1..4} |},
        Filter_index = {| i : Int • i ∈ {1..3} |}
      object
        P[i:Chan_index] : class channel event : Event,  eoe : Unit end
      value
        test : Filter_index → Event → Bool
        test(i) ≡
          case i of
            1 → test1,
            2 → test2,
            3 → test3
          end
      variable result : Event* := ⟨⟩
      value
        filt : Filter_index → in any out any Unit
        filt(i) ≡
          local variable terminated : Bool := false
          in
            while ∼terminated do
              P[i].eoe? ; P[i+1].eoe!() ; terminated := true
              ⌷
              let e = P[i].event? in
                if test(i)(e) then P[i+1].event!e end
              end
            end
          end,
        send : Event* → in any out any Unit
        send(el) ≡
          local variable to_do : Event* := el in
            while to_do ≠ ⟨⟩ do
              P[1].event!hd to_do ; to_do := tl to_do
            end ;
            P[1].eoe!()
          end,
        collect : Unit → in any out any write result Unit
        collect() ≡
          local variable terminated : Bool := false
          in
```

```
          while ~terminated do
            P[4].eoe? ; terminated := true
            ⌷
            let e = P[4].event? in result := result ⁀ ⟨e⟩ end
          end
        end
    in
      (send(el) ∥ ∥{filt(i) | i : Filter_index} ∥ collect()) ; result
    end
```

Page 206

```
scheme A_BOUNDED_SET(E : ELEM_BOUND) =
  hide set_of, Set_of_Set in
  class
    type Set, Set_of_Set = {| s : Elem-set • card s ≤ E.Bound |}
    value
      /* generators */
      empty : Set,
      add : E.Elem × Set ~→ Set,
      remove : E.Elem × Set → Set,
      /* hidden_observer */
      set_of : Set → Set_of_Set,
      /* derived observers */
      is_in : E.Elem × Set → Bool
      is_in(e, s) ≡ e ∈ set_of(s),
      can_add : E.Elem × Set → Bool
      can_add(e, s) ≡ e ∈ set_of(s) ∨ card set_of(s) < E.bound
    axiom
      [ set_of_empty ] set_of(empty) ≡ {},
      [ set_of_add ]
         ∀ s : Set, e : E.Elem • set_of(add(e, s)) ≡ {e} ∪ set_of(s)
      pre can_add(e, s),
      [ set_of_remove ]
         ∀ s : Set, e : E.Elem • set_of(remove(e, s)) ≡ set_of(s) \ {e},
      [ add_defined ]
         ∀ s : Set, e : E.Elem • add(e, s) post true pre can_add(e, s)
  end
```

Page 211

Exercise 1

```
scheme LIST_SET(E : ELEM) =
  hide no_duplicates in
  class
    type Set = {| el : E.Elem* • no_duplicates(el) |}
    value
      empty : Set = ⟨⟩,
      add : E.Elem × Set → Set
```

```
      add(e, s) ≡ if is_in(e, s) then s else ⟨e⟩^s end,
      remove : E.Elem × Set → Set
      remove(e, s) ≡
        case s of
          ⟨⟩ → ⟨⟩,
          ⟨h⟩^t →
            if h = e then t else ⟨h⟩^remove(e, t) end
        end,
      is_in : E.Elem × Set → Bool
      is_in(e, s) ≡
        case s of
          ⟨⟩ → false,
          ⟨h⟩^t → e = h ∨ is_in(e, t)
        end,
      no_duplicates : E.Elem* → Bool
      no_duplicates(el) ≡ card elems el = len el
  end
```

To show that this implements *ABS_SET*, we need to show a conservative extension defining *set_of* implements *ABS_SET*, i.e. that

```
class object E : ELEM end ⊢
  extend LIST_SET(E) with
  hide set_of in
  class
    value set_of : Set → E.Elem-set = elems
  end ⪯
    ABS_SET(E)
```

The extension is clearly conservative. We need to show the axioms of *ABS_SET* are true in the extension of *LIST_SET*. Take *set_of_add* as an example. Ignoring quantification, we need to show

```
  set_of(add(e, s)) ≡ {e} ∪ set_of(s)
```

i.e. (unfolding)

```
  elems (if is_in(e, s) then s else ⟨e⟩^s end) ≡ {e} ∪ elems s
```

i.e.

```
  if is_in(e, s) then elems s else elems (⟨e⟩^s) end ≡ {e} ∪ elems s
```

i.e.

```
  if is_in(e, s) then elems s else {e} ∪ elems s end ≡ {e} ∪ elems s
```

We show separately (to show that *is_in* is implemented) that

```
  is_in(e, s) ≡ e ∈ elems s
```

and the result follows from the properties of set union.

Page 211

Exercise 2

```
scheme ABS_BAG(E : ELEM) =
  class
    type Bag
    value
      /* generators */
      empty : Bag,
```

```
      add : E.Elem × Bag → Bag,
      remove : E.Elem × Bag -~-> Bag,
      /∗ observer ∗/
      count : E.Elem × Bag → Nat
    axiom
      [count_empty] ∀ e : E.Elem • count(e, empty) ≡ 0,
      [count_add]
        ∀ b : Bag, e, e′ : E.Elem •
          count(e′, add(e, b)) ≡ if e = e′ then count(e′, b) + 1 else count(e′, b) end,
      [count_remove]
        ∀ b : Bag, e, e′ : E.Elem •
          count(e′, remove(e, b)) ≡ if e = e′ then count(e′, b) − 1 else count(e′, b) end
          pre count(e, b) > 0,
      [remove_defined] ∀ b : Bag, e : E.Elem • remove(e, b) post true pre count(e, b) > 0
  end
```

To show *LIST_BAG* implements *ABS_BAG* we need to show the axioms of *ABS_BAG* are true in *LIST_BAG*. Take *count_add* as an example. Ignoring quantification, we need to show

```
  count(e′, add(e, b)) ≡ if e = e′ then count(e′, b) + 1 else count(e′, b) end
```

i.e. (unfolding)

```
  if ⟨e⟩^b = ⟨⟩ then 0
  else if hd ⟨e⟩^b = e′ then count(e′, tl ⟨e⟩^b) + 1 else count(e′, tl ⟨e⟩^b) end
  end ≡
    if e = e′ then count(e′, b) + 1 else count(e′, b) end
```

and this follows immediately.

```
scheme MAP_BAG(E : ELEM) =
  hide Nat1 in
  class
    type Nat1 = {| n : Nat • n > 0 |},   Bag = E.Elem -m-> Nat1
    value
      empty : Bag = [ ],
      add : E.Elem × Bag → Bag
      add(e, b) ≡ if e ∈ dom b then b † [e ↦ b(e) + 1] else b † [e ↦ 1] end,
      remove : E.Elem × Bag -~-> Bag
      remove(e, b) ≡
        if count(e, b) = 1 then b \ {e} else b † [e ↦ b(e) − 1] end
      pre count(e, b) > 0,
      count : E.Elem × Bag → Nat
      count(e, b) ≡
        if e ∈ dom b then b(e) else 0 end
  end
```

To show *MAP_BAG* implements *ABS_BAG* we need to show the axioms of *ABS_BAG* are true in *MAP_BAG*. Take *count_remove* as an example. Ignoring quantification, and assuming that *count(e, b)* is strictly positive, we need to show

```
  count(e′, remove(e, b)) ≡ if e = e′ then count(e′, b) − 1 else count(e′, b) end
```

i.e. (unfolding)

```
  if e′ ∈ dom remove(e, b) then remove(e, b)(e′) else 0 end ≡
    if e = e′ then count(e′, b) − 1 else count(e′, b) end
```

Consider two cases:

- count(e, b) = 1

remove(e, b) is now $b \setminus \{e\}$ and our goal reduces to

$$\textbf{if } e' \in (\textbf{dom } b) \setminus \{e\} \textbf{ then } (b \setminus \{e\})(e') \textbf{ else } 0 \textbf{ end} \equiv$$
$$\quad \textbf{if } e = e' \textbf{ then } count(e', b) - 1 \textbf{ else } count(e', b) \textbf{ end}$$

If $e = e'$, this reduces to

$0 \equiv count(e', b) - 1$
i.e. (from the assumption $e = e'$)
$0 \equiv count(e, b) - 1$
i.e. (from the assumption $count(e, b) = 1$)
$0 \equiv 1 - 1$

and we are finished. If $e \neq e'$, we have instead

$$\textbf{if } e' \in \textbf{dom } b \textbf{ then } b(e') \textbf{ else } 0 \textbf{ end} \equiv count(e', b)$$

and this follows immediately on unfolding *count*.

- $count(e, b) > 1$

remove(e, b) is now $b \dagger [\, e \mapsto b(e) - 1\,]$ and our goal reduces to

$$\textbf{if } e' \in \textbf{dom } b \textbf{ then } (b \dagger [\, e \mapsto b(e) - 1\,])(e') \textbf{ else } 0 \textbf{ end} \equiv$$
$$\quad \textbf{if } e = e' \textbf{ then } count(e', b) - 1 \textbf{ else } count(e', b) \textbf{ end}$$

and again the result follows when we consider the two cases $e = e'$ and $e \neq e'$.

Page 214

value
eql : Bag × Bag → **Bool**
eql(b1, b2) ≡ (∀ e : E.Elem • count(e, b1) = count(e, b2))

No congruence axioms would be needed, since *A_BAG* follows the standard style and *count* is the only non-derived observer.

Page 215

The definition with the postcondition does implement the original definition with only a signature, since the postcondition ensures that f terminates with a strictly positive result, which therefore must certainly be a **Nat**.

If the second definition were only a signature, implementation would not follow. The developed f could terminate with a negative result.

Page 218

value
isin_range : R.Elem × Map → **Bool**
isin_range(r, m) ≡ (∃ d : D.Elem • is_in(d, m) ∧ apply(d, m) = r)

Page 219

1. A signature-axiom definition for *isin_range* could be

$$
\begin{array}{l}
\textbf{value} \\
\quad \mathrm{isin\_range} : \mathrm{R.Elem} \times \mathrm{Map} \to \textbf{Bool} \\
\textbf{axiom} \\
\quad [\,\mathrm{isin\_range\_empty}\,]\ \forall\, \mathrm{r} : \mathrm{R.Elem} \bullet \sim \mathrm{isin\_range(r, empty)}, \\
\quad [\,\mathrm{isin\_range\_add}\,] \\
\qquad \forall\, \mathrm{m} : \mathrm{Map},\ \mathrm{d} : \mathrm{D.Elem},\ \mathrm{r, r'} : \mathrm{R.Elem} \bullet \\
\qquad\quad \mathrm{isin\_range(r', add(d, r, m))} \equiv \\
\qquad\qquad \mathrm{r = r'} \vee (\exists\, \mathrm{d'} : \mathrm{D.Elem} \bullet \mathrm{d \neq d'} \wedge \mathrm{is\_in(d', m)} \wedge \mathrm{apply(d', m) = r'}), \\
\quad [\,\mathrm{isin\_range\_remove}\,] \\
\qquad \forall\, \mathrm{m} : \mathrm{Map},\ \mathrm{d} : \mathrm{D.Elem},\ \mathrm{r} : \mathrm{R.Elem} \bullet \\
\qquad\quad \mathrm{isin\_range(r', remove(d, m))} \equiv \\
\qquad\qquad (\exists\, \mathrm{d'} : \mathrm{D.Elem} \bullet \mathrm{d \neq d'} \wedge \mathrm{is\_in(d', m)} \wedge \mathrm{apply(d', m) = r'})
\end{array}
$$

2. The definition of *isin_range* as a derived observer is shorter and clearer than the three axioms. This would be true even if the axioms were not as complicated as these are.

3. We show that *isin_range_add* follows from the definition of *isin_range*. Ignoring quantification, we need to show

$$
\begin{array}{l}
\quad \mathrm{isin\_range(r', add(d, r, m))} \equiv \\
\qquad \mathrm{r = r'} \vee (\exists\, \mathrm{d'} : \mathrm{D.Elem} \bullet \mathrm{d \neq d'} \wedge \mathrm{is\_in(d', m)} \wedge \mathrm{apply(d', m) = r'}) \\
\text{i.e. (unfolding)} \\
\quad (\exists\, \mathrm{d'} : \mathrm{D.Elem} \bullet \mathrm{is\_in(d', add(d, r, m))} \wedge \mathrm{apply(d', add(d, r, m)) = r'}) \equiv \\
\qquad \mathrm{r = r'} \vee (\exists\, \mathrm{d'} : \mathrm{D.Elem} \bullet \mathrm{d \neq d'} \wedge \mathrm{is\_in(d', m)} \wedge \mathrm{apply(d', m) = r'}) \\
\text{i.e. (from the other axioms)} \\
\quad (\exists\, \mathrm{d'} : \mathrm{D.Elem} \bullet (\mathrm{d = d'} \vee \mathrm{is\_in(d', m)}) \wedge \textbf{if}\ \mathrm{d = d'}\ \textbf{then}\ \mathrm{r}\ \textbf{else}\ \mathrm{apply(d', m)}\ \textbf{end} = \mathrm{r'}) \equiv \\
\qquad \mathrm{r = r'} \vee (\exists\, \mathrm{d'} : \mathrm{D.Elem} \bullet \mathrm{d \neq d'} \wedge \mathrm{is\_in(d', m)} \wedge \mathrm{apply(d', m) = r'})
\end{array}
$$

We now consider two cases, $r = r'$ and $r \neq r'$.

In the first case, the right-hand side reduces to true. The left-hand side reduces to

$$(\exists\, \mathrm{d'} : \mathrm{D.Elem} \bullet (\mathrm{d = d'} \vee \mathrm{is\_in(d', m)}) \wedge \textbf{if}\ \mathrm{d = d'}\ \textbf{then true else}\ \mathrm{apply(d', m) = r'}\ \textbf{end})$$

and this can be shown to be true by using d as a "witness".

In the second case, the restriction on the left-hand side reduces to

$$
\begin{array}{l}
\quad (\mathrm{d = d'} \vee \mathrm{is\_in(d', m)}) \wedge \textbf{if}\ \mathrm{d = d'}\ \textbf{then false else}\ \mathrm{apply(d', m) = r'}\ \textbf{end} \\
\text{i.e.} \\
\quad \textbf{if}\ \mathrm{d = d'}\ \textbf{then false else}\ \mathrm{is\_in(d', m)} \wedge \mathrm{apply(d', m) = r'}\ \textbf{end} \\
\text{i.e.} \\
\quad \mathrm{d \neq d'} \wedge \mathrm{is\_in(d', m)} \wedge \mathrm{apply(d', m) = r'}
\end{array}
$$

and so the goal reduces to

$$
\begin{array}{l}
\quad (\exists\, \mathrm{d'} : \mathrm{D.Elem} \bullet \mathrm{d \neq d'} \wedge \mathrm{is\_in(d', m)} \wedge \mathrm{apply(d', m) = r'}) \equiv \\
\qquad \textbf{false} \vee (\exists\, \mathrm{d'} : \mathrm{D.Elem} \bullet \mathrm{d \neq d'} \wedge \mathrm{is\_in(d', m)} \wedge \mathrm{apply(d', m) = r'})
\end{array}
$$

which is obviously true.

4. A suitable induction axiom is

$$
\begin{array}{l}
[\,\mathrm{Map\_induction}\,] \\
\quad \forall\, \mathrm{p} : \mathrm{Map} \to \textbf{Bool} \bullet \\
\qquad (\mathrm{p(empty)} \wedge \\
\qquad\ (\forall\, \mathrm{d} : \mathrm{D.Elem},\ \mathrm{r} : \mathrm{R.Elem},\ \mathrm{m} : \mathrm{Map} \bullet \\
\qquad\qquad \mathrm{p(m)} \Rightarrow (\mathrm{p(add(d, r, m))} \wedge \mathrm{p(remove(d, m))}))) \Rightarrow \\
\qquad (\forall\, \mathrm{m} : \mathrm{Map} \bullet \mathrm{p(m)})
\end{array}
$$

Page 220

```
scheme A_STACK (E : ELEM) =
  hide head, tail in
  extend
    use Stack for List, push for cons in
    A_LIST(E)
  with
  class
    value
      pop : Stack ~→ E.Elem × Stack
      pop(s) ≡ (head(s), tail(s)) pre ∼is_empty(s)
  end
```

This is more useful specification of a stack if we already have an implementation of *A_LIST* and not of *A_STACK*, but otherwise rather less clear than the original.

Page 229

Exercise 1

1. We assume the types *Index* and *Machine_state*.

```
type
  Map = Index →m Machine_state
value
  reset_all : Machine_state → Map
  reset_all(s) ≡ [ i ↦ s | i : Index ]
```

2. We assume the specification of *MACHINE*.

```
object O[ i : Index ] : MACHINE
value
  reset_all : Machine_state → in any out any Unit
  reset_all(s) ≡ ||{O[ i ].reset(s) | i : Index}
```

3. Our version executes the individual resets in parallel. We are assuming that the *MACHINE*s in the array do not communicate with each other and do not share any imperative modules. Then the parallel composition of the *reset*s is equivalent to an arbitrary interleaving of the individual *reset*s, and all interleavings are equivalent.

4. We assume there are only finitely many values in *Index*, and we redefine *reset_all* as follows

```
value
  reset_all : Machine_state → in any out any Index-set
  reset_all(s) ≡
    local
      variable result : Index-set := {}
      value
        number_indices : Nat = card {i | i : Index},
        collect : Nat → in reset_res write result Unit
        collect(n) ≡
          if n = 0 then skip
          else
```

```
            let (i,r) = reset_res? in
              if r = busy then result := result ∪ {i} end ; collect(n − 1) end
          end
      in
        (‖{reset_res!(i, O[i].reset(s)) | i : Index} ‖ collect(number_indices)) ; result
      end
```

This definition shows that it is not necessary to make the machines "self conscious".

Page 230

Exercise 2

1. **value**
   no_cycles : Tree → **read any Bool**
   no_cycles(t) ≡
     if t = 0 **then true else** left(t) < t ∧ right(t) < t **end**
2. **value**
   eql : Tree × Tree → **Bool**
   eql(t1, t2) ≡
     if t1 = 0 **then** t2 = 0
     elsif t2 = 0 **then false**
     else val(t1) = val(t2) ∧ eql(left(t1), left(t2)) ∧ eql(right(t1), right(t2)) **end**

   Note that we could write the body of *eql* as

   t1 = t2 ∨ ...

   where ... is the body in the version given.

3. We need to show (ignoring quantification)

   □ is_tree(l) ∧ is_tree(r) ⇒
     (**let** t = node(l,e,r) **in** is_tree(t) **end** ≡ **let** t = node(l,e,r) **in true end**)

   i.e.
   □ is_tree(l) ∧ is_tree(r) ⇒
     (**let** t =
       next := next + 1 ;
       let t = next **in** S[t].left := l ; S[t].val := e ; S[t].right := r ; t **end**
     in is_tree(t) **end** ≡ **let** t = ... **in true end**)

   Suppose the current value of *next* is *k* (which is arbitrary since the goal is quantified by □). Then this reduces to

   □ is_tree(l) ∧ is_tree(r) ⇒
     (next := k+1 ; S[k+1].left := l ; S[k+1].val := e ; S[k+1].right := r ; is_tree(k+1) ≡
       next := k+1 ; S[k+1].left := l ; S[k+1].val := e ; S[k+1].right := r ; **true**)

   The expression *is_tree(k+1)* expands to

   k+1 = 0 ∨
   k+1 ≤ next ∧ no_cycles(k+1) ∧ is_tree(S[k+1].left) ∧ is_tree(S[k+1].right)

   i.e.
   k+1 ≤ k+1 ∧ no_cycles(k+1) ∧ is_tree(l) ∧ is_tree(r)

   i.e. (from the implication)
   no_cycles(k+1)

   i.e.

if k+1 = 0 **then** **true** **else** left(k+1) < k+1 ∧ right(k+1) < k+1 **end**
i.e.
l < k+1 ∧ r < k+1

and this follows since, for example,

is_tree(l) ⇒ (l = 0 ∨ l ≤ next)
i.e.
is_tree(l) ⇒ (l = 0 ∨ l ≤ k)

So *l* (and similarly *r*) are strictly less than *k+1*.

4. Ignoring quantification, we need to show

   □ **let** l′ = node(l,e,r) **in** eql(left(l′),l) **end** ≡ node(l,e,r) ; **true**

   (Axioms are implicitly quantified by □.)

   We again assume a current value for *next* of *k*. Then we have

   □ next := k+1 ; S[k+1].left := l ; S[k+1].val := e ; S[k+1].right := r ; eql(left(k+1),l) ≡
     next := k+1 ; S[k+1].left := l ; S[k+1].val := e ; S[k+1].right := r ; **true**

   In the call of *eql*

   left(k+1) ≡ S[k+1].left ≡ l

   and so the goal reduces to showing

   □ eql(l,l)

   The definition of *eql* is clearly reflexive, and we are finished.

Page 230

Exercise 3

value
  deq : Queue $\stackrel{\sim}{\to}$ **write any** E.Elem × Queue
  deq(q) ≡
    let mk_Queue(f, b) = q **in**
      let f′ = S[f].back, r = S[f].val **in**
        F.add(f) ; (r, mk_Queue(f′,b))
      end
    end
  pre is_queue(q) ∧ ∼is_empty(q),
  no_cycles : Index × Index → **read any Bool**
  no_cycles(i, j) ≡
    local
      value
        forward_reachables : Index → **read any** Index-**set**
        forward_reachables(i) ≡
          if i = 0 **then** {0} **else** {i} ∪ forward_reachables(S[i].front) **end**,
        backward_reachables : Index → **read any** Index-**set**
        backward_reachables(i) ≡
          if i = 0 **then** {0} **else** {i} ∪ backward_reachables(S[i].back) **end**
    in
      j ∉ backward_reachables(i) ∧ i ∉ forward_reachables(j)

```
    end,
  connected : Index × Index → read any Bool
  connected(f, b) ≡
    f = b ∨ (f ≠ 0 ∧ connected(S[f].back, b) ∧ b ≠ 0 ∧ connected(f, S[b].front))
```

Page 230

Exercise 4

- We remove the objects X and F and add a variable *free*:

  variable free : Index

- We add an *init* function:

  ```
  value
    init : Unit → write any Unit
    init() ≡
      for i in ⟨1..E.bound−1⟩ do S[i].back := i+1 end ;
      S[E.bound].back := 0 ;
      free := 1
  ```

- We replace the body of the axiom *initially_all_available* with

  initialise ; init() ≡ **initialise**

- We add (and hide) the following definitions:

  ```
  value
    can_select : Unit → read any Bool
    can_select() ≡ free ≠ 0,
    select : Unit ~→ write any Index
    select() ≡ let r = free in free := S[free].back ; r end
    pre can_select(),
    add : Index ~→ write any Unit
    add(i) ≡ S[i].back := free ; free := i
    pre i ≥ 1
  ```

- We replace calls of functions *F.can_select* etc. with *can_select* etc.

Page 232

```
scheme SORT0(E: LINEAR_ORDER) =
  hide is_permutation, is_ordered in
  class
    value
      sort : E.Elem* → E.Elem*
      sort(el) as el′ post is_permutation(el, el′) ∧ is_ordered(el′),
      is_permutation : E.Elem* × E.Elem* → Bool
      is_permutation(el, el′) ≡
        local
          value
            count : E.Elem × E.Elem* → Nat
            count(e, el) ≡
```

```
            case el of
              ⟨⟩ → 0,
              ⟨h⟩^t → if h = e then count(e, t) + 1 else count(e, t) end
            end
        in
          (∀ e : E.Elem • count(e,el) = count(e, el′))
        end,
      is_ordered : E.Elem* → Bool
      is_ordered(el) ≡
        local
          value
            leq : E.Elem × E.Elem* → Bool
            leq(e, el) ≡ (∀ e′ : E.Elem • e′ ∈ elems el ⇒ E.leq(e, e′))
        in
          (∀ el1, el2 : E.Elem*, e : E.Elem •
            el = el1^⟨e⟩^el2 ⇒ leq(e, el2))
        end
  end
```

where *leq* defined in *is_ordered* is is like the overloaded $\leq$ defined in the solution to the exercise from page 192 of the book.

```
scheme SORT1(E: LINEAR_ORDER) =
  hide remove_min in
  class
    value
      sort : E.Elem* → E.Elem*
      sort(el) ≡
        if el = ⟨⟩ then el
        else let (e, el′) = remove_min(el) in ⟨e⟩^sort(el′) end end,
      remove_min : E.Elem* → E.Elem × E.Elem*
      remove_min(el) ≡ ...
      pre el ≠ ⟨⟩
  end
```

where the body of *remove_min* is as given on page 187 of the book, but with *E.leq* replacing $\leq$.

The implementation relation to show that *SORT1* implements *SORT0* is

```
class object E : LINEAR_ORDER end ⊢
  extend SORT1(E) with
  hide is_permutation, is_ordered in
  class
     is_permutation : E.Elem* × E.Elem* → Bool
     is_permutation(el, el′) ≡ ...,
     is_ordered : E.Elem* → Bool
     is_ordered(el) ≡ ...
  end ≼
    SORT0(E)
```

where the definitions of *is_permutation* and *is_ordered* are as in *SORT0*.

To show the extension used is conservative we need to show it is consistent with subtypes, which reduces to showing there exist total functions *is_permutation* and *is_ordered* satisfying their specifications. This is obviously the case, but note it also involves checking the existence of the total functions in the local expressions.

To show implementation we need to show the *is_permutation* and *is_ordered* properties of *sort*.

For *is_permutation* we note that *count* has the property

$$\forall \text{ el1, el2} : \text{E.Elem}^* \bullet \text{count}(\text{el1}^\frown\text{el2}) \equiv \text{count}(\text{el1}) + \text{count}(\text{el2}) \quad (1)$$

This can be proved by induction on *el1*.

We also know that *remove_min* satisfies a postcondition:

$$\begin{array}{l} \text{remove\_min(el)} \ \textbf{as} \ (\text{r, el}') \ \textbf{post} \\ \quad \text{leq(r, el}') \wedge \\ \quad (\exists \text{ l1, l2} : \text{E.Elem}^* \bullet \text{l1}^\frown\langle \text{r}\rangle^\frown\text{l2} = \text{el} \wedge \text{l1}^\frown\text{l2} = \text{el}') \\ \quad \textbf{pre} \ \text{el} \neq \langle\rangle \end{array} \quad (2)$$

(2) corresponds to the postcondition for *remove_min* from page 187 of the book. From (2), the definition of *sort*, and the property (1) of *count*, we can see that

$$\forall \text{ el} : \text{E.Elem}^* \bullet \text{is\_permutation(el, sort(el))}$$

This can be proved by induction on *el*.

Finally, from the definition of *sort* and the first conjunct of (2), we can see that

$$\forall \text{ el} : \text{E.Elem}^* \bullet \text{is\_ordered(sort(el))}$$

This can be proved by induction on *el*, using the properties of *leq* corresponding to those for the overloaded $\leq$ defined in the solution to the exercise from page 192 of the book.

This completes the justification that *SORT1* implements *SORT0*.

It is worth noting that in such justifications it is often more useful to have properties like (1) for *count*, the postcondition for *remove_min*, and the properties for *leq*, than it is to have their definitions.

If *is_permutation* and *is_ordered* are defined in *LIST_FUNS* then we would write *SORT0* as

```
scheme SORT0(E: LINEAR_ORDER) =
  hide LF in
  class
    object LF : LIST_FUNS(E)
    value
      sort : E.Elem* → E.Elem*
      sort(el) as el′ post LF.is_permutation(el, el′) ∧ LF.is_ordered(el′)
  end
```

SORT1 would be defined as above, assuming *remove_min* and *leq* were not also defined in *LIST_FUNS*. Then the implementation relation would be written

```
class object E : LINEAR_ORDER end ⊢
  extend SORT1(E) with
  hide LF in
  class
     object LF : LIST_FUNS(E)
  end ≼
    SORT0(E)
```

Conservativeness of the extension will depend on the consistency of *LIST_FUNS*.

Page 234

We can define a general module for giving concurrent access to a variable:

```
scheme C_VAR(E : ELEM) =
  hide CH, v, main in
  class
    object CH : class channel put, get : E.Elem end
    variable v : E.Elem
```

```
    value
      /* initial */
      init : Unit → in any out any write any Unit
      init() ≡ main(),
      /* main */
      main : Unit → in any out any write any Unit
      main() ≡
      while true do
        CH.get!v
        ⌷
        v := CH.put?
      end,
      /* interface functions */
      put : E.Elem → in any out any Unit
      put(e) ≡ CH.put!e,
      get : Unit → in any out any E.Elem
      get() ≡ CH.get?
  end
```

Then *C_BUTTON1* could be defined by

```
scheme C_BUTTON1 =
  use check for get in
  hide put in
  extend C_VAR(T{Button_state for Elem}) with
  class
    value
      push : Unit → in any out any Unit
      push() ≡ put(T.lit),
      clear : Unit → in any out any Unit
      clear() ≡ put(T.clear)
  end
```

This is the same as the original *C_BUTTON1* apart from the name of the (hidden) main process.

A similar construction could be used for *C_DOOR1*, except that it has acknowledgement channels to model in particular the fact that the doors are actually shut when *close* terminates. So *C_VAR* is not quite sufficient. We could include in *C_VAR* acknowledgement channels and an extra parameter to decide whether these channels are used, but it seems clumsy and (unless we are careful in translation) likely to generate run-time overheads.

Page 236

For a definition of *filter*, see the local function *filt* in the solution to the exercise from page 183 of the book.

```
value
  remove_all : T → T* → T*
  remove_all(e) ≡
    let neq = (λ x : T • x ≠ e) in filter(neq) end,
  remove_duplicates : T* → T*
  remove_duplicates(seq) ≡
    case seq of
      ⟨⟩ → ⟨⟩,
      ⟨h⟩^t → ⟨h⟩^remove_duplicates(remove_all(h)(t))
```

end

Page 237

value
factorial : **Int** → **Nat**
factorial(i) ≡ **if** i ≤ 0 **then** 1 **else** i∗factorial(i−1) **end**

Page 239

value
filter : (T → **Bool**) → T* → T*
filter(p)(seq) **as** seq′ **post**
(∀ e : Elem • count(e, seq′) = **if** ∼p(e) **then** 0 **else** count(e, seq) **end**) ∧
(∀ seq1′, seq2′ : T*, e1, e2 : T •
seq′ = seq1′⌢⟨e1,e2⟩⌢seq2′ ⇒
(∃ seq1, seq2, seq3 : T* •
seq = seq1⌢⟨e1⟩⌢seq3⌢⟨e2⟩⌢seq2 ∧
(∀ e : T • e ∈ **elems** seq3 ⇒ ∼p(e))))

This uses the *count* function we defined in the solution to the exercise from page 232 of the book.

The first conjunct of the postcondition says that the elements not satisfying *p* are removed. The second conjunct says the relative order of elements in the result is preserved. To allow the relative order to be changed we would simply remove the second conjunct.

Page 243

scheme A_SAFE_MAP(D : ELEM, R : ELEM) =
hide apply **in**
extend A_MAP(E) **with**
class
type Apply_result == not_found | found(res : R.Elem)
value
safe_apply : D.Elem × Map → Apply_result
safe_apply(d, m) ≡
if is_in(d, m) **then** found(apply(d, m)) **else** not_found **end**
end

scheme I_SAFE_MAP(D : ELEM, R : ELEM) =
hide apply **in**
extend I_MAP(E) **with**
class
type Apply_result == not_found | found(res : R.Elem)
value
safe_apply : D.Elem → **read any** Apply_result
safe_apply(d) ≡
if is_in(d) **then** found(apply(d)) **else** not_found **end**
end

We cannot so easily change the concurrent version by extension because if we call the concurrent *is_in* and then the concurrent *apply* there is no guarantee that the result of the first call will hold at the time of the second. So in the concurrent case we would adapt *C_MAP* by

- changing the definition of the object I to

  object I : I_SAFE_MAP(D, R)

- adding the definitions

  type Apply_result = I.Apply_result
  value
  not_found : Apply_result = I.not_found,
  found : R.Elem → Apply_result = I.found,
  res : Apply_result $\xrightarrow{\sim}$ R.Elem = I.res

- changing the signature of *apply* to

  safe_apply : D.Elem → **in any out any** Apply_result

- changing the final axiom to

  [safe_apply_ax]
  ∀ d : D.Elem, test : Apply_result $\xrightarrow{\sim}$ **Unit** •
  main() ∦ test(safe_apply(d)) ≡ **let** r = I.safe_apply(d) **in** main() ∦ test(r) **end**

Page 256

In S, we know there is at least one value a in T, but this may be the only one. We know nothing about the value of a.

Hence the following declarations would *not* extend S conservatively:

variable v : T := 0
value b : T • b ≠ a
value f : **Int** → T f(x) **as** r **post** r > a

The first asserts that 0 is in T. The second and third each assert the existence of a value in T that is different from a.

The other declarations in the exercise would extend S conservatively.

Chapter 4

Page 286

⌞s = {}⌟
set_equality :
⌞∀ i : **Int** • (i ∈ s) = (i ∈ {})⌟
all_assumption_inf :
⌞(i ∈ s) = (i ∈ {})⌟
empty :
⌞(**false**) = (i ∈ {})⌟
isin_empty :
⌞(**false**) = (**false**)⌟
equality_annihilation :
⌞**true**⌟
qed

Page 289

$\llcorner \mathrm{I} \vdash \forall\ \mathrm{i} : \mathbf{Int} \bullet \sim \mathrm{member}(\mathrm{i}, \langle\rangle) \lrcorner$
class_assumption_inf :
$\llcorner \forall\ \mathrm{i} : \mathbf{Int} \bullet \sim \mathrm{member}(\mathrm{i}, \langle\rangle) \lrcorner$
all_assumption_inf :
$\llcorner \sim \mathrm{member}(\mathrm{i}, \langle\rangle) \lrcorner$
member_def :
$\llcorner \sim (\langle\rangle \neq \langle\rangle \wedge (\mathrm{i} = \mathbf{hd}\ \langle\rangle \vee \mathrm{member}(\mathrm{i}, \mathbf{tl}\ \langle\rangle))) \lrcorner$
inequality_expansion :
$\llcorner \sim (\sim (\langle\rangle = \langle\rangle) \wedge (\mathrm{i} = \mathbf{hd}\ \langle\rangle \vee \mathrm{member}(\mathrm{i}, \mathbf{tl}\ \langle\rangle))) \lrcorner$
equality_annihilation :
$\llcorner \sim (\sim (\mathbf{true}) \wedge (\mathrm{i} = \mathbf{hd}\ \langle\rangle \vee \mathrm{member}(\mathrm{i}, \mathbf{tl}\ \langle\rangle))) \lrcorner$
not_true :
$\llcorner \sim (\mathbf{false} \wedge (\mathrm{i} = \mathbf{hd}\ \langle\rangle \vee \mathrm{member}(\mathrm{i}, \mathbf{tl}\ \langle\rangle))) \lrcorner$
and_expansion :
$\llcorner \sim \mathbf{if\ false\ then}\ \mathrm{i} = \mathbf{hd}\ \langle\rangle \vee \mathrm{member}(\mathrm{i}, \mathbf{tl}\ \langle\rangle)\ \mathbf{else\ false\ end} \lrcorner$
if_false :
$\llcorner \sim \mathbf{false} \lrcorner$
not_false :
$\llcorner \mathbf{true} \lrcorner$
qed

Page 293

Exercise 1

$\llcorner \forall\ \mathrm{p} : \mathbf{Nat} \times \mathbf{Nat} \bullet \mathbf{let}\ (\mathrm{x,y}) = \mathrm{p}\ \mathbf{in}\ \mathrm{x} \geq 0 \wedge \mathrm{y} \geq 0\ \mathbf{end} \lrcorner$
all_name_change :
$\llcorner \forall\ (\mathrm{r,s}) : \mathbf{Nat} \times \mathbf{Nat} \bullet \mathbf{let}\ (\mathrm{x,y}) = (\mathrm{r,s})\ \mathbf{in}\ \mathrm{x} \geq 0 \wedge \mathrm{y} \geq 0\ \mathbf{end} \lrcorner$
let_absorption4 :
$\llcorner \forall\ (\mathrm{r,s}) : \mathbf{Nat} \times \mathbf{Nat} \bullet\ \mathrm{r} \geq 0 \wedge \mathrm{s} \geq 0 \lrcorner$
and we now have the same example as in the text.

Page 293

Exercise 2

$\llcorner \sim (\exists\ \mathrm{x} : \{|\ \mathrm{I} : \mathbf{Int} \bullet \mathbf{false}\ |\} \bullet \mathbf{true}) \lrcorner$
exists_subtype :
$\llcorner \sim (\exists\ \mathrm{x} : \mathbf{Int} \bullet \mathbf{false} \wedge \mathbf{true}) \lrcorner$
and_expansion :
$\llcorner \sim (\exists\ \mathrm{x} : \mathbf{Int} \bullet \mathbf{if\ false\ then\ true\ else\ false\ end}) \lrcorner$
if_false :
$\llcorner \sim (\exists\ \mathrm{x} : \mathbf{Int} \bullet \mathbf{false}) \lrcorner$
exists_expansion :
$\llcorner \sim (\sim (\forall\ \mathrm{x} : \mathbf{Int} \bullet \sim (\mathbf{false} \equiv \mathbf{true}))) \lrcorner$
is_true :

⌞∼ (∼ (∀ x : **Int** • ∼ (**false**)))⌟
not_false :
⌞∼ (∼ (∀ x : **Int** • **true**))⌟
not_not :
⌞∀ x : **Int** • **true**⌟
all_assumption_inf :
⌞**true**⌟
qed

Page 294

⌞∀ t : Tree • t = empty ∨ (∃ l, r : Tree, el : Elem • t = node(l, el, r))⌟
substitution1:
⌞∀ t : Tree • (λ t : Tree • t = empty ∨ (∃ l, r : Tree, el : Elem • t = node(l, el, r)))(t)⌟
since
⌞(λ t : Tree • t = empty ∨ (∃ l, r : Tree, el : Elem • t = node(l, el, r)))(t) ≡
t = empty ∨ (∃ l, r : Tree, el : Elem • t = node(l, el, r))⌟
lambda_application, let_absorption4, is_annihilation, **qed**
end
Tree_induction :
⌞**true**⌟
since

- ⌞(λ t : Tree • t = empty ∨ (∃ l, r : Tree, el : Elem • t = node(l, el, r)))(empty)⌟
  lambda_application, let_absorption4 :
  ⌞empty = empty ∨ (∃ l, r : Tree, el : Elem • empty = node(l, el, r))⌟
  simplify, **qed**
- ⌞∀ t1 : Tree, elem : Elem, t2 : Tree •
  (λ t : Tree • t = empty ∨ (∃ l, r : Tree, el : Elem • t = node(l, el, r)))(t1) ∧
  (λ t : Tree • t = empty ∨ (∃ l, r : Tree, el : Elem • t = node(l, el, r)))(t2) ⇒
  (λ t : Tree • t = empty ∨
  (∃ l, r : Tree, el : Elem • t = node(l, el, r)))(node(t1, elem, t2))⌟
  lambda_application, let_absorption4, lambda_application, let_absorption4,
  lambda_application, let_absorption4 :
  ⌞∀ t1 : Tree, elem : Elem, t2 : Tree •
  (t1 = empty ∨ (∃ l, r : Tree, el : Elem • t1 = node(l, el, r))) ∧
  (t2 = empty ∨ (∃ l, r : Tree, el : Elem • t2 = node(l, el, r))) ⇒
  node(t1, elem, t2) = empty ∨
  (∃ l, r : Tree, el : Elem • node(t1, elem, t2) = node(l, el, r))⌟
  all_assumption_inf :
  ⌞(t1 = empty ∨ (∃ l, r : Tree, el : Elem • t1 = node(l, el, r))) ∧
  (t2 = empty ∨ (∃ l, r : Tree, el : Elem • t2 = node(l, el, r))) ⇒
  node(t1, elem, t2) = empty ∨
  (∃ l, r : Tree, el : Elem • node(t1, elem, t2) = node(l, el, r))⌟
  imply_deduction_inf :
  [ind_hyp] (t1 = empty ∨ (∃ l, r : Tree, el : Elem • t1 = node(l, el, r))) ∧
  (t2 = empty ∨ (∃ l, r : Tree, el : Elem • t2 = node(l, el, r))) ⊢
  ⌞node(t1, elem, t2) = empty ∨
  (∃ l, r : Tree, el : Elem • node(t1, elem, t2) = node(l, el, r))⌟
  node_empty :

```
      ⌞false ∨ (∃ l, r : Tree, el : Elem • node(t1, elem, t2) = node(l, el, r))⌟
    simplify :
      ⌞∃ l, r : Tree, el : Elem • node(t1, elem, t2) = node(l, el, r)⌟
    exists_introduction_inf :
      ⌞let (l, r, el) = (t1, t2, elem) in node(t1, elem, t2) = node(l, el, r) end⌟
    let_absorption4, simplify, qed
  end
qed
```

Page 299

The outline of the proof is shown. The details in each of the four innermost case branches are just properties of arithmetic; only the first is completed.

```
 ⌞∀ x, y : Int • abs (x + y) ≤ abs x + abs y⌟
all_assumption_inf, abs_int_expansion, abs_int_expansion, abs_int_expansion:
 ⌞if x + y ≥ 0 then x + y else 0 − (x + y) end ≤
    if x ≥ 0 then x else 0 − x end + if y ≥ 0 then y else 0 − y end⌟
two_cases_inf :
• [x_non_neg] x ≥ 0 ⊢
   ⌞if x + y ≥ 0 then x + y else 0 − (x + y) end ≤
      if x ≥ 0 then x else 0 − x end + if y ≥ 0 then y else 0 − y end⌟
  simplify :
    ⌞if x + y ≥ 0 then x + y else 0 − (x + y) end ≤ x + if y ≥ 0 then y else 0 − y end⌟
  two_cases_inf :
  • [y_non_neg] y ≥ 0 ⊢
     ⌞if x + y ≥ 0 then x + y else 0 − (x + y) end ≤ x + if y ≥ 0 then y else 0 − y end⌟
    simplify :
     ⌞if x + y ≥ 0 then x + y else 0 − (x + y) end ≤ x + y⌟
    if_infix_op :
     ⌞if x + y ≥ 0 then x + y ≤ x + y else 0 − (x + y) ≤ x + y end⌟
    simplify :
     ⌞if x + y ≥ 0 then true else 2 ∗ (x + y) ≥ 0 end⌟
    geq_int_transitivity :
     ⌞if true then true else 2 ∗ (x + y) ≥ 0 end⌟
    since
       ⌞(x + y ≥ x ∧ x ≥ 0)⌟
      simplify, qed
    end
    simplify, qed
  • [y_neg] ∼ (y ≥ 0) ⊢
     ⌞if x + y ≥ 0 then x + y else 0 − (x + y) end ≤ x + if y ≥ 0 then y else 0 − y end⌟
    simplify :
     ⌞if x + y ≥ 0 then x + y else 0 − (x + y) end ≤ x − y⌟
    /∗ which can be reduced to true ∗/ qed
• [x_neg] ∼ (x ≥ 0) ⊢
   ⌞if x + y ≥ 0 then x + y else 0 − (x + y) end ≤
      if x ≥ 0 then x else 0 − x end + if y ≥ 0 then y else 0 − y end⌟
  simplify :
```

$\llcorner$**if** x + y ≥ 0 **then** x + y **else** 0 − (x + y) **end** ≤
0 − x + **if** y ≥ 0 **then** y **else** 0 − y **end**$\lrcorner$
two_cases_inf :
- [y_non_neg] y ≥ 0 ⊢
  $\llcorner$**if** x + y ≥ 0 **then** x + y **else** 0 − (x + y) **end** ≤
  0 − x + **if** y ≥ 0 **then** y **else** 0 − y **end**$\lrcorner$
  simplify :
  $\llcorner$**if** x + y ≥ 0 **then** x + y **else** 0 − (x + y) **end** ≤ 0 − x + y$\lrcorner$
  /∗ which can be reduced to true ∗/ **qed**
- [y_neg] ∼ (y ≥ 0) ⊢
  $\llcorner$**if** x + y ≥ 0 **then** x + y **else** 0 − (x + y) **end** ≤
  0 − x + **if** y ≥ 0 **then** y **else** 0 − y **end**$\lrcorner$
  simplify :
  $\llcorner$**if** x + y ≥ 0 **then** x + y **else** 0 − (x + y) **end** ≤ 0 − x − y$\lrcorner$
  /∗ which can be reduced to true ∗/ **qed**

Page 302

$\llcorner$∀ el : **Int**∗ • **len** el ≥ **card** (**elems** el)$\lrcorner$
all_list_left_induction :
$\llcorner$**true**$\lrcorner$
since
- $\llcorner$**len** ⟨⟩ ≥ **card** (**elems** ⟨⟩)$\lrcorner$
  simplify, **qed**
- $\llcorner$∀ el : **Int**∗ • ∀ e : **Int** • **len** el ≥ **card** (**elems** el) ⇒
  len (⟨e⟩ ⌢ el) ≥ **card** (**elems** (⟨e⟩ ⌢ el))$\lrcorner$
  all_assumption_inf, all_assumption_inf :
  $\llcorner$**len** el ≥ **card** (**elems** el) ⇒ **len** (⟨e⟩ ⌢ el) ≥ **card** (**elems** (⟨e⟩ ⌢ el))$\lrcorner$
  imply_deduction_inf :
  [ind_hyp] **len** el ≥ **card** (**elems** el) ⊢ $\llcorner$**len** (⟨e⟩ ⌢ el) ≥ **card** (**elems** (⟨e⟩ ⌢ el))$\lrcorner$
  simplify :
  $\llcorner$1 + **len** el ≥ **card** ({e} ∪ **elems** el)$\lrcorner$
  card_union :
  $\llcorner$1 + **len** el ≥ **card** {e} + **card** **elems** el − **card** ({e} ∩ **elems** el)$\lrcorner$
  since
  - $\llcorner$**card** **elems** el **post** **true**$\lrcorner$
    simplify, **qed**
  - $\llcorner$**card** {e} **post** **true**$\lrcorner$
    simplify, **qed**

  end
  card_singleton :
  $\llcorner$1 + **len** el ≥ 1 + **card** **elems** el − **card** ({e} ∩ **elems** el)$\lrcorner$
  simplify :
  $\llcorner$**len** el + **if** e ∈ **elems** el **then** 1 **else** 0 **end** ≥ **card** **elems** el$\lrcorner$
  geq_int_transitivity :
  $\llcorner$**true**$\lrcorner$
  since
  $\llcorner$**len** el + **if** e ∈ **elems** el **then** 1 **else** 0 **end** ≥ **len** el ∧

```
              len el ≥ card elems el⌟
            simplify, qed
          end
        qed
    end
qed
```

Page 309

```
  ⌞k > 4⌟
follows
  from
    [greater_transitivity]   ⌞∀ x, y, z : Int • x > y ∧ y > z ⇒ x > z⌟,
    [k_greater_5]            ⌞k > 5⌟
      ⌞∀ x, y, z : Int • x > y ∧ y > z ⇒ x > z⌟,   ⌞k > 5⌟
    all_instantiation_inf ⇒
      ⌞let (x, y, z) = (k, 5, 4) in x > y ∧ y > z ⇒ x > z end⌟,   ⌞k > 5⌟
    let_absorption4 ⇒
      ⌞k > 5 ∧ 5 > 4 ⇒ k > 4⌟,   ⌞k > 5⌟
    simplify ⇒
      ⌞k > 5 ∧ true ⇒ k > 4⌟,   ⌞k > 5⌟
    and_true ⇒
      ⌞k > 5 ⇒ k > 4⌟,   ⌞k > 5⌟
    reorder ⇒
      ⌞k > 5⌟,   ⌞k > 5 ⇒ k > 4⌟
    imply_modus_ponens_inf ⇒
      ⌞k > 4⌟
    qed
end
```

Page 311

```
1.  ⌞(∃ f : T → Bool • f(x) ∧ ∼ f(y)) ⇒ x ≠ y⌟
    exists_implies :
      ⌞∀ f : T → Bool • f(x) ∧ ∼ f(y) ⇒ x ≠ y⌟
    all_assumption_inf :
      ⌞f(x) ∧ ∼ f(y) ⇒ x ≠ y⌟
    imply_deduction_inf :
    [f_hyp] f(x) ∧ ∼ f(y) ⊢ ⌞x ≠ y⌟
    inequality_expansion :
      ⌞∼ (x = y)⌟
    contradiction_inf2 :
    [x_eq_y] x = y ⊢ ⌞false⌟
    follows
      from
        [f_hyp]   ⌞f(x) ∧ ∼ f(y)⌟
          ⌞f(x) ∧ ∼ f(y)⌟
        x_eq_y ⇒
```

⌞f(x) ∧ ∼ f(x)⌟
and_annihilation ⇒
⌞**false**⌟
qed
end

2. ⌞x = y ⇒ ∼ (∃ f : T → **Bool** • f(x) ∧ ∼ f(y))⌟
imply_deduction_inf :
[x_eq_y] x = y ⊢ ⌞∼ (∃ f : T → **Bool** • f(x) ∧ ∼ f(y))⌟
exists_expansion :
⌞∼ ∼ (∀ f : T → **Bool** • ∼ (f(x) ∧ ∼ f(y) ≡ **true**))⌟
simplify :
⌞∀ f : T → **Bool** • ∼ (f(x) ∧ ∼ f(y))⌟
all_assumption_inf :
⌞∼ (f(x) ∧ ∼ f(y))⌟
x_eq_y :
⌞∼ (f(x) ∧ ∼ f(x))⌟
and_annihilation :
⌞∼ **false**⌟
not_false :
⌞**true**⌟
qed

Page 318

The effect of the sequence of the three assignments is to interchange the values in x and y.

The justification starts by introducing an □ and then introducing arbitrary initial assignments a and b for x and y.

⌞x := x + y ; y := x − y ; x := x − y **post** x > y **pre** y > x⌟
always_elimination_inf :
⌞□ x := x + y ; y := x − y ; x := x − y **post** x > y **pre** y > x⌟
always_post_application1, all_assumption_inf, assignment_sequence_propagation, always_application1, all_assumption_inf, assignment_sequence_propagation, assignment_sequence_propagation :
⌞□ y := b ; x := a ; (x := a + b ; y := a + b − b ; x := a + b − y **post** x > y **pre** b > a) ≡
y := b ; x := a ; **true**⌟
sequence_post :
⌞□ y := b ; x := a ; (**true**) ≡ y := b ; x := a ; **true**⌟
since

- ⌞x := a + b **post** b > a **pre** b > a⌟
  simplify, **qed**
- ⌞y := a + b − b ; x := a + b − y **post** x > y **pre** b > a⌟
  simplify, assignment_sequence_propagation :
  ⌞y := a ; x := a + b − a **post** x > a **pre** b > a⌟
  sequence_post :
  ⌞**true**⌟
  since
  - ⌞y := a **post** b > a **pre** b > a⌟
    simplify, **qed**
  - ⌞x := a + b − a **post** x > a **pre** b > a⌟

assignment_post_propagation :
⌊x := a + b − a **post** a + b − a > a **pre** b > a⌋
simplify, **qed**
end
qed
end
simplify, **qed**

Page 324

⌊∀ x, y : **Int** • divide(x, y) **post** x = R + y ∗ Q ∧ y > R **pre** y > 0⌋
pre_post_deduction_inf :
[y_pos] y > 0 ⊢
⌊divide(x, y) **post** x = R + y ∗ Q ∧ y > R⌋
divide_def, opt_pre1 :
⌊(R := x ; Q := 0) ; **while** y ≤ R **do** R := R − y ; Q := Q + 1 **end**
post x = R + y ∗ Q ∧ y > R **pre true**⌋
/∗ split before while, using invariant x = R + y ∗ Q as intermediate
condition ∗/
sequence_post :
⌊**true**⌋
since
- ⌊R := x ; Q := 0 **post** x = R + y ∗ Q **pre true**⌋
  assignment_sequence_propagation :
  ⌊R := x ; Q := 0 **post** x = x + y ∗ Q **pre true**⌋
  /∗ post condition does not mention R,
  so split using precondition for intermediate condition ∗/
  sequence_post :
  ⌊**true**⌋
  since
  - ⌊R := x **post true pre true**⌋
    simplify, **qed**
  - ⌊Q := 0 **post** x = x + y ∗ Q **pre true**⌋
    assignment_post_propagation, simplify, **qed**

  end
  qed
- ⌊**while** y ≤ R **do** R := R − y ; Q := Q + 1 **end**
  post x = R + y ∗ Q ∧ y > R **pre** x = R + y ∗ Q⌋
  /∗ get into appropriate form for while_post ∗/
  greater_int_expansion :
  ⌊**while** y ≤ R **do** R := R − y ; Q := Q + 1 **end**
  post x = R + y ∗ Q ∧ ∼ (y ≤ R) **pre** x = R + y ∗ Q⌋
  /∗ use R−y as decreasing measure ∗/
  while_post :
  ⌊**true**⌋
  since
  - ⌊**while** (x = R + y ∗ Q ≡ **true**) ∧ y ≤ R **do**
    R := R − y ; Q := Q + 1
    end post true⌋

```
simplify :
  ⌞while x = R + y ∗ Q ∧ R ≥ y do
      R := R − y ; Q := 1 + Q
    end post true⌟
while_convergence :
  ⌞true⌟
  since
    • ⌞□ (x = R + y ∗ Q ∧ R ≥ y post true) ∧
            (R − y post true)⌟
      simplify, qed
    • ⌞□ R − y < 0 ⇒ ∼ (x = R + y ∗ Q ∧ R ≥ y)⌟
      /∗ condition that postcondition true on termination.
         Introduce initial value for R and simplify ∗/
      is_true, always_application1, assignment_sequence_propagation,
      not_and, simplify, qed
    • ⌞□ let b = R − y in
            (R := R − y ; Q := 1 + Q) ; b > R − y end ≡
              (R := R − y ; Q := 1 + Q) ; true
            pre x = R + y ∗ Q ∧ R ≥ y⌟
      /∗ condition that loop body reduces measure.
         Introduce initial value for R and simplify ∗/
      always_application2, all_assumption_inf,
      assignment_sequence_propagation, assignment_sequence_propagation,
      assignment_sequence_propagation, simplify,
      assignment_sequence_propagation, simplify, qed
  end
qed
• ⌞□ R := R − y ; Q := Q + 1
      post x = R + y ∗ Q pre x = R + y ∗ Q ∧ y ≤ R⌟
/∗ condition that loop body maintains invariant.
   Introduce initial value for R and simplify ∗/
always_post_application1, all_assumption_inf,
assignment_sequence_propagation, assignment_sequence_propagation :
  ⌞□ R := r ;
     (R := r − y ; Q := Q + 1
      post x = r − y + y ∗ Q pre x = r + y ∗ Q ∧ y ≤ r) ≡
       R := r ; true⌟
/∗ post condition no longer mentions R,
   so split using as intermediate condition the precondition ∗/
sequence_post :
  ⌞□ R := r ; (true) ≡ R := r ; true⌟
  since
    • ⌞R := r − y
          post x = r + y ∗ Q ∧ y ≤ r pre x = r + y ∗ Q ∧ y ≤ r⌟
      simplify, qed
    • ⌞Q := Q + 1
          post x = r − y + y ∗ Q pre x = r + y ∗ Q ∧ y ≤ r⌟
      /∗ introduce initial value for q and simplify ∗/
      always_elimination_inf, always_post_application1,
```

```
                all_assumption_inf, assignment_sequence_propagation,
                assignment_post_propagation, simplify, qed
            end
          i.e. :
            ⌞□ R := r ; (true) ≡ R := r ; true⌟
          simplify, qed
      end
    qed
  end
qed
```

Page 331

We have

parallel_ints(c!1, v:=c?)
≡
parallel_ints(v:=c?, c!1)
≡
parallel_ints(**let** b = c? **in** v:=b **end**, c!1; **skip**)
≡
let b = 1 **in** v:=b || **skip** **end**
≡
v:=1

and

parallel_exts(c!1, v:=c?)
≡
parallel_exts(c!1; **skip**, v:=c?)
≡
c!1 ; (**skip** || v:=c?)
≡
c!1 ; v:=c?

Similarly

parallel_exts(v:=c?, c!1) ≡ v:=c? ; c!1

So

c!1 || v:=c?
≡
parallel_expand(c!1, v:=c?)
≡
(**parallel_exts**(c!1, v:=c?) [] **parallel_exts**(v:=c?, c!1) ⌈⌉ **parallel_ints**(c!1, v:=c?)) ⌈⌉
parallel_ints(c!1, v:=c?)
≡
(c!1 ; v:=c? [] v:=c? ; c!1 ⌈⌉ v:=1) ⌈⌉ v:=1

as required.

Page 333

Exercise 1

```
⌞stack_p(⟨e⟩^st) ╫ pop() ≡ stack_p(st)⌟
stack_p_def, pop_def :
 ⌞(is_empty_c ! (⟨e⟩^st = ⟨⟩) ; stack_p(⟨e⟩^st)
   ⌷ let b = push_c? in stack_p(⟨b⟩^⟨e⟩^st) end
   ⌷ if ∼ (⟨e⟩^st = ⟨⟩) then pop_c? ; stack_p(tl ⟨e⟩^st) else stop end
   ╫
   pop_c ! ()) ≡ stack_p(st)⌟
simplify :
 ⌞(is_empty_c ! false ; stack_p(⟨e⟩^st)
   ⌷ let b = push_c? in stack_p(⟨b⟩^⟨e⟩^st) end
   ⌷ pop_c? ; stack_p(st))
   ╫
   pop_c ! () ≡ stack_p(st)⌟
interlock_expansion, simplify, qed
```

Page 333

Exercise 2

The definition of *stack_p* is

```
stack_p : Stack → in any out any Unit
stack_p(st) ≡
  local variable v : Stack := st in
    while true do
      is_empty_c ! (v = ⟨⟩)
      ⌷
      let b = push_c? in v := ⟨b⟩ ^ v end
      ⌷
      if ∼ (v = ⟨⟩) then pop_c? ; v := tl v else stop end
      ⌷
      if ∼ (v = ⟨⟩) then top_c ! hd v else stop end
    end
  end
```

To justify the condition (1) we unfold *stack_p* on the left, unwind the while loop with *while_unwinding*, introduce the initial assignment with *local_variable* and propagate it. Then we need to commute the assignments to *v* with the inputs and outputs in each choice. Our goal is then

```
⌞(local variable v : Stack := st in
    is_empty_c ! (st = ⟨⟩ ; v := st) ; W ⌷
    let x = push_c? in let b = v := st ; x in v := ⟨b⟩ ^ st end end ; W ⌷
    if ∼ (st = ⟨⟩) then pop_c? ; v := tl st else v := st ; stop end ; W ⌷
    if ∼ (st = ⟨⟩) then top_c ! hd st ; v := st else v := st ; stop end ; W
  end ╫ push_c!e) ╫ pop_c?  ≡  stack_p(st)⌟
```

where *W* is the **while true do** ... **end** loop from the definition of *stack_p*.

For simplicity we will take the case where $st = \langle\rangle$. This, plus *local_interlock*, allows us to use *interlock_expansion* and after simplifying we get

```
⌞local variable v : Stack := ⟨⟩ in
    v := ⟨e⟩ ; W
  end ╫  pop_c?  ≡  stack_p(⟨⟩)⌟
```

Now we repeat the original tactic of unwinding the loop etc. and we get

$\llcorner$**local variable** v : Stack := $\langle\rangle$ **in**
v := $\langle\rangle$; W
end $\equiv$ stack_p($\langle\rangle$)$\lrcorner$

and the justification is completed by unfolding on the right and applying *local_variable*.

Chapter 5

Page 379

Translating the scheme *C_DOORS1* into C requires making certain assumptions about the translation of *TYPES* and *T*. Hence, we first provide the translation of *T*, which is just an instantiation of *TYPES*, into C.

```
/* T.h */

#ifndef T_H
#define T_H

typedef unsigned char Bool;

typedef int Floor;
typedef int Lower_Floor;
typedef int Upper_Floor;

enum _Door_State {opened, shut}; /* no overloading, open renamed to opened */
enum _Button_State {lit, clear};
enum _Direction {up, down};
enum _Movement {halted, moving};

typedef enum _Door_State Door_State;
typedef enum _Button_State Button_State;
typedef enum _Direction Direction;
typedef enum _Movement Movement;

struct _Requirement { Bool here, after, before; };
typedef struct _Requirement *Requirement;

void mk_Requirement(Bool here, Bool after, Bool before, Requirement r);
Bool here(Requirement r);
Bool after(Requirement r);
Bool before(Requirement r);

#define MIN_FLOOR 1
#define MAX_FLOOR 10
#define FLOOR_NO (MAX_FLOOR - MIN_FLOOR + 1)
const Floor min_floor;
const Floor max_floor;
Bool is_floor(Floor f);

Floor next_floor(Direction d, Floor f);
Bool is_next_floor(Direction d, Floor f);
```

```
Direction invert(Direction d);

#endif


/* T.c */

#include "T.h"

const Floor min_floor = MIN_FLOOR;
const Floor max_floor = MAX_FLOOR;

Bool is_floor(Floor f)
{ return f >= min_floor && f <= max_floor; }

void mk_Requirement(Bool here, Bool after, Bool before, Requirement r)
{
  r->here = here;
  r->after = after;
  r->before = before;
}

Bool here(Requirement r)
{ return r->here; }

Bool after(Requirement r)
{ return r->after; }

Bool before(Requirement r)
{ return r->before; }

Floor next_floor(Direction d, Floor f)
{
  if (d == up)
    return f+1;
  else
    return f-1;
}

Bool is_next_floor(Direction d, Floor f)
{
  if (d == up)
    return f < max_floor;
  else
    return f > min_floor;
}

Direction invert(Direction d)
{
  if (d == up)
    return down;
```

```
  else
    return up;
}
```

In C_DOORS1 we have decided to use only two sockets and use different values (of type Message) sent via the sockets to distinguish between the different communications. Moreover, error situations are explicitly reported to the user.

```
/* C_DOORS1.h */

#ifndef C_DOORS1_H
#define C_DOORS1_H

#include "T.h"

/* initial */
void init();

/* generators */
void open_door(Floor f); /* no overloading, open renamed to open_door */
void close_door(Floor f); /* no overloading, close renamed to close_door */

/* observer */
typedef Door_State T_Door_State;
T_Door_State door_state(Floor f);

#endif


/* C_DOORS1.c */

#include <sys/types.h>
#include <sys/socket.h>
#include <stdio.h>
#include "C_DOORS1.h"

enum _Message
  {open_msg, close_msg, open_ack_msg, close_ack_msg, door_state_msg};

typedef enum _Message Message;

int sockets[FLOOR_NO][2];

void perrorexit(char *s)
{
  perror(s);
  exit(1);
}

Message message_input(Floor f, int s)
{ Message m;
```

```
  if (read(sockets[f - min_floor][s], &m, sizeof(Message)) < 0)
    perrorexit("message_input");
  return m;
}

void message_output(Floor f, int s, Message m)
{
  if (write(sockets[f - min_floor][s], &m, sizeof(Message)) < 0)
    perrorexit("message_output");
}

Door_State state_input(Floor f, int s)
{ Door_State state;

  if (read(sockets[f - min_floor][s], &state, sizeof(Door_State)) < 0)
    perrorexit("state_input");
  return state;
}

void state_output(Floor f, int s, Door_State state)
{
  if (write(sockets[f - min_floor][s], &state, sizeof(Door_State)) < 0)
    perrorexit("state_output");
}

void door_proc(Floor f)
{ Door_State state;
  Message m;

  while (1)
  { m = message_input(f, 0);
    switch (m)
    { case open_msg:
        state = opened;
        message_output(f, 0, open_ack_msg);
        break;
      case close_msg:
        state = shut;
        message_output(f, 0, close_ack_msg);
        break;
      case door_state_msg:
        state_output(f, 0, state);
        break;
      default:
        printf("Unknown message %i\n", m);
        exit(1);
    }
  }
}

void init_door(Floor f)
```

```
{ int child;

  if (socketpair(AF_UNIX, SOCK_STREAM, 0, sockets[f - min_floor]) < 0)
    perrorexit("init_door");

  if ((child = fork()) == -1)
    perror("fork");
  else if (child)
  { /* This is the parent */
    close(sockets[f - min_floor][0]);
  }
  else
  { /* This is the child */
     close(sockets[f - min_floor][1]);
     door_proc(f);
  }
}

/* initial */
void init()
{ Floor f;

  for (f = min_floor; f <= max_floor; f++)
    init_door(f);
}

/* generators */
void open_door(Floor f)
{
  message_output(f, 1, open_msg);
  if (message_input(f, 1) != open_ack_msg)
  { fprintf(stderr, "Not receiving open_ack\n");
    exit(1);
  }
}

void close_door(Floor f)
{
  message_output(f, 1, close_msg);
  if (message_input(f, 1) != close_ack_msg)
  { fprintf(stderr, "Not receiving close_ack\n");
    exit(1);
  }
}

/* observer */
T_Door_State door_state(Floor f)
{
  message_output(f, 1, door_state_msg);
  return state_input(f, 1);
}
```

Page 405

Exercise 1

The package specification of `C_BUTTONS1` only needs to declare the three subprograms corresponding to the non-hidden functions.

```
with T; use T;
generic
package C_BUTTONS1 is
  -- initial
  procedure init;

  -- generators
  procedure clear(f : in T.Floor);

  -- observer
  function check(d : in T.Direction; f : in T.Floor)
              return T.Requirement;
end C_BUTTONS1;
```

The scheme *C_BUTTONS1* contains three object arrays of *C_BUTTON1*. We will translate them following the same approach as that used for *C_DOORS1*, namely to declare only one package for each array. The main change in the package declaration of `LB`, `UB` and `DB` is that each function gets the floor number as an extra parameter.

```
with Int_Package; use Int_Package;
package body C_BUTTONS1 is
  package LB is
    -- initial
    procedure init(f : in T.Floor);
    -- generators
    procedure push(f : in T.Floor);
    procedure clear(f : in T.Floor);
     -- observer
    function check(f : in T.Floor) return T.Button_state;
  end LB;

  package UB is
    -- initial
    procedure init(f : in T.Lower_floor);
    -- generators
    procedure push(f : in T.Lower_floor);
    procedure clear(f : in T.Lower_floor);
     -- observer
    function check(f : in T.Lower_floor) return T.Button_state;
  end UB;

  package DB is
    -- initial
    procedure init(f : in T.Upper_floor);
    -- generators
    procedure push(f : in T.Upper_floor);
    procedure clear(f : in T.Upper_floor);
```

```
   -- observer
  function check(f : in T.Upper_floor) return T.Button_state;
end DB;

package body LB is
  button_vars : array(T.Floor) of T.Button_state;

  task type button_type is
    entry CH_push;
    entry CH_clear;
    entry CH_check(s : out T.Button_state);
    entry id(f : in T.Floor);
  end button_type;

  type acc_button_type is access button_type;
  acc_button_vars : array(T.Floor) of acc_button_type;

  task body button_type is
    fl : T.Floor;
  begin
    accept id(f : in T.Floor) do
      fl := f;
    end id;
    loop
      select
        accept CH_push; button_vars(fl) := T.lit;
      or
        accept CH_clear; button_vars(fl) := T.clear;
      or
        accept CH_check(s : out T.Button_state) do
          s := button_vars(fl);
        end CH_check;
      end select;
    end loop;
  end button_type;

  procedure button(f : in T.Floor) is
  begin
    acc_button_vars(f) := new button_type;
    acc_button_vars(f).id(f);
  end button;

  -- initial
  procedure init(f : in T.Floor) is
  begin
    button(f);
  end init;

  -- generators
  procedure push(f : in T.Floor) is
  begin
```

```
    acc_button_vars(f).CH_push;
  end push;

  procedure clear(f : in T.Floor) is
  begin
    acc_button_vars(f).CH_clear;
  end clear;

   -- observer
  function check(f : in T.Floor) return T.Button_state is
    bs : T.Button_state;
  begin
    acc_button_vars(f).CH_check(bs);
    return bs;
  end check;
end LB;

package body UB is
  button_vars : array(T.Lower_floor) of T.Button_state;

  task type button_type is
    entry CH_push;
    entry CH_clear;
    entry CH_check(s : out T.Button_state);
    entry id(f : in T.Lower_floor);
  end button_type;

  type acc_button_type is access button_type;
  acc_button_vars : array(T.Lower_floor) of acc_button_type;

  task body button_type is
    fl : T.Lower_floor;
  begin
    accept id(f : in T.Lower_floor) do
      fl := f;
    end id;
    loop
      select
        accept CH_push; button_vars(fl) := T.lit;
      or
        accept CH_clear; button_vars(fl) := T.clear;
      or
        accept CH_check(s : out T.Button_state) do
          s := button_vars(fl);
        end CH_check;
      end select;
    end loop;
  end button_type;

  procedure button(f : in T.Lower_floor) is
  begin
```

```
      acc_button_vars(f) := new button_type;
      acc_button_vars(f).id(f);
    end button;

    -- initial
    procedure init(f : in T.Lower_floor) is
    begin
      button(f);
    end init;

    -- generators
    procedure push(f : in T.Lower_floor) is
    begin
      acc_button_vars(f).CH_push;
    end push;

    procedure clear(f : in T.Lower_floor) is
    begin
      acc_button_vars(f).CH_clear;
    end clear;

     -- observer
    function check(f : in T.Lower_floor) return T.Button_state is
      bs : T.Button_state;
    begin
      acc_button_vars(f).CH_check(bs);
      return bs;
    end check;
  end UB;

  package body DB is
    button_vars : array(T.Upper_floor) of T.Button_state;

    task type button_type is
      entry CH_push;
      entry CH_clear;
      entry CH_check(s : out T.Button_state);
      entry id(f : in T.Upper_floor);
    end button_type;

    type acc_button_type is access button_type;
    acc_button_vars : array(T.Upper_floor) of acc_button_type;

    task body button_type is
      fl : T.Upper_floor;
    begin
      accept id(f : in T.Upper_floor) do
        fl := f;
      end id;
      loop
        select
```

```
        accept CH_push; button_vars(fl) := T.lit;
      or
        accept CH_clear; button_vars(fl) := T.clear;
      or
        accept CH_check(s : out T.Button_state) do
          s := button_vars(fl);
        end CH_check;
      end select;
    end loop;
  end button_type;

  procedure button(f : in T.Upper_floor) is
  begin
    acc_button_vars(f) := new button_type;
    acc_button_vars(f).id(f);
  end button;

  -- initial
  procedure init(f : in T.Upper_floor) is
  begin
    button(f);
  end init;

  -- generators
  procedure push(f : in T.Upper_floor) is
  begin
    acc_button_vars(f).CH_push;
  end push;

  procedure clear(f : in T.Upper_floor) is
  begin
    acc_button_vars(f).CH_clear;
  end clear;

   -- observer
  function check(f : in T.Upper_floor) return T.Button_state is
    bs : T.Button_state;
  begin
    acc_button_vars(f).CH_check(bs);
    return bs;
  end check;
end DB;


-- initial
procedure init is
begin
  for f in T.Floor loop
    LB.init(f);
  end loop;
  for f in T.Lower_floor loop
```

```
    UB.init(f);
  end loop;
  for f in T.Upper_floor loop
    DB.init(f);
  end loop;
end init;

-- generators
procedure clear(f : in T.Floor) is
begin
  LB.clear(f);
  if f < T.max_floor then UB.clear(f); end if;
  if f > T.min_floor then DB.clear(f); end if;
end clear;

-- observers
function required_beyond(d : in T.Direction; f : in T.Floor)
           return Boolean;

function required_here(d : in T.Direction; f : in T.Floor)
           return Boolean is
begin
  return
    LB.check(f) = T.lit or else
     (d = T.up and then
      ((f < T.max_floor and then UB.check(f) = T.lit) or else
       (f > T.min_floor and then DB.check(f) = T.lit and then
        not required_beyond(d,f))))
     or else
      (d = T.down and then
        ((f > T.min_floor and then DB.check(f) = T.lit) or else
         (f < T.max_floor and then UB.check(f) = T.lit and then
          not required_beyond(d,f))));
end required_here;

function required_beyond(d : in T.Direction; f : in T.Floor)
           return Boolean is
  f1 : T.Floor := T.next_floor(d,f);
begin
  if T.is_next_floor(d, f) then
    declare
      f1 : T.Floor := T.next_floor(d,f);
    begin
      return T.is_floor(f1) and then
             (required_here(d,f1) or else required_beyond(d,f1));
    end;
  else
    return false;
  end if;
end required_beyond;
```

```
  function check(d : in T.Direction; f : in T.Floor)
            return T.Requirement is
  begin
    return T.mk_Requirement(required_here(d,f),
                            required_beyond(d,f),
                            required_beyond(T.invert(d),f));
  end check;
end C_BUTTONS1;
```