Proposed solutions

1 Logic

1. The final answers are given below.

   (a) It holds always.

   (b) It holds always.

   (c) It does not hold if a ≡ **chaos**.
   It holds if a ≡ **true** or a ≡ **false**.

   (d) It holds always.

   (e) It does not hold if a ≡ **chaos**.
   It holds if a ≡ **true** or a ≡ **false**.

   (f) It holds always.

   (g) It holds always.

   (h) It does not hold if a ≡ **chaos**.
   It holds if a ≡ **true** or a ≡ **false**.

   (i) It holds always.

2. (a) **if true then false else chaos end** ≡ **false** cf. (1)

   (b) **if** a **then** ∼(a ≡ **chaos**) **else false end** ≡ cf. (3)
   if a **then** ∼(**true** ≡ **chaos**) **else false end** ≡
   if a **then true else false end** ≡ cf. (3)
   if a **then** a **else** a **end** ≡
   a

3. (a) is true (for given i, choose j = -i)

   (b) is false

   (c) is false

4. ∼ (∃ i : **Int** •(∀ j : **Int** • i ≥ j))

   alternatively:

   ∀ i : **Int** •(∃ j : **Int** • j ≥ i)

5. is_even : **Nat** → **Bool**
   is_even(n) ≡ (∃ m : **Nat** • n = 2 * m)

   alternatively:

   is_even : **Nat** → **Bool**
   is_even(n) ≡ n \ 2 = 0

2 Products

1. (a) **type**
   Complex = **Real** × **Real**

   (b) **value**
   zero : Complex = (0.0,0.0)

   (c) **value**
   c : Complex • **let** (x,y) = c **in** x = y **end**

   (d) **value**
   add : Complex × Complex → Complex
   add((x_1,y_1),(x_2,y_2)) ≡ ($x_1 + x_2$, $y_1 + y_2$),

   mult : Complex × Complex → Complex
   mult((x_1,y_1),(x_2,y_2)) ≡ ($x_1 * x_2 - y_1 * y_2$, $x_1 * y_2 + y_1 * x_2$)

   (e) **value**
   f : Complex → Complex
   f(c_1) **as** c_2 **post** $c_1 \neq c_2$

3 Functions

1. (a) + : **Int** × **Int** → **Int**
   + : **Real** × **Real** → **Real**

   (b) ∗ : **Int** × **Int** → **Int**
   ∗ : **Real** × **Real** → **Real**

   (c) \ : **Int** × **Int** $\stackrel{\sim}{\to}$ **Int**

   (d) ↑ : **Int** × **Int** $\stackrel{\sim}{\to}$ **Int**
   ↑ : **Real** × **Real** $\stackrel{\sim}{\to}$ **Real**

2. (a) **value**
   max : **Int** × **Int** → **Int**
   max(i,j) ≡ **if** i ≥ j **then** i **else** j **end**

   (b) **value**
   max : **Int** × **Int** → **Int**
   max(i,j) **as** y
   post (y = i ∧ y ≥ j) ∨ (y = j ∧ y ≥ i)

   (c) **value**
   max : **Int** × **Int** → **Int**
   axiom
   ∀ i,j : **Int** • max(i,j) ≡ **if** i ≥ j **then** i **else** j **end**

3. approx_sqrt : **Real** × **Real** $\stackrel{\sim}{\to}$ **Real**
   approx_sqrt(x, eps) **as** s
   post s ≤ x ↑ 0.5 ∧ x ↑ 0.5 < s+eps
   pre x ≥ 0.0 ∧ eps > 0.0

4 Sets

1. {1,3,5,7,9}
   {n | n : **Nat** • n ∈ {0 .. 10} ∧ is_odd(n)}

   value
   is_odd : **Nat** → **Bool**
   is_odd(n) ≡ n\2 ≠ 0

2. (a) True

   (b) True

3. **value**
   dunion : (Elem-**set**)-**set** → Elem-**set**
   dunion(ss) ≡ { e | e : Elem • ∃ s : Elem-**set** • e ∈ s ∧ s ∈ ss }

5 Lists

1. **type** Elem

   value
   length : Elem* → **Int**
   length(l) ≡ **if** l = ⟨⟩ **then** 0 **else** 1 + length(**tl** l) **end**,

   rev : Elem* → Elem*
   rev(l) ≡ **if** l = ⟨⟩ **then** ⟨⟩ **else** rev(**tl** l) ⁀ ⟨**hd** l⟩ **end**,

   drev : (Elem*)* → (Elem*)*
   drev(ll) ≡ **if** ll = ⟨⟩ **then** ⟨⟩ **else** drev(**tl** ll) ⁀ ⟨rev(**hd** ll)⟩ **end**

   alternative definitions:

   length : Elem* → **Int**
   length(l) ≡ **card** (**inds** l),

   length : Elem* → **Int**
   length(l) ≡
   case l **of**
   ⟨⟩ → 0,
   ⟨i⟩⁀lr → length(lr) + 1
   end,

   rev : Elem* → Elem*
   rev(l) ≡ ⟨l(**len** l − i + 1) | i **in** ⟨1 .. **len** l⟩⟩,

   rev : Elem* → Elem*
   rev(l) ≡
   case l **of**
   ⟨⟩ → ⟨⟩,
   ⟨i⟩⁀lr → rev(lr) ⁀ ⟨**hd** l⟩
   end,

   drev : (Elem*)* → (Elem*)*

```
    drev(ll) ≡ ⟨rev(ll(len ll − i + 1)) | i in ⟨1 .. len ll⟩⟩,

    drev : (Elem*)* → (Elem*)*
    drev(ll) ≡
        case ll of
            ⟨⟩ → ⟨⟩,
            ⟨l⟩⌢llr → drev(llr) ⌢ ⟨drev(hd ll)⟩
        end
```

2. **type** N1 = {| n : **Nat** • n ≥ 1 |}

```
value
    pascal : N1 → (N1*)*
    pascal(n) ≡
        if n = 1 then
            ⟨⟨1⟩⟩
        else
            let p = pascal(n − 1) in
                p ⌢ ⟨⟨1⟩ ⌢ ⟨ p(n − 1)(i − 1) + p(n − 1)(i) | i in ⟨ 2 .. n − 1 ⟩ ⟩ ⌢ ⟨1⟩⟩
            end
        end
```

alternatively:

```
value
    pascal : N1 → (N1*)*
    pascal(n) ≡ ⟨ aux_pascal(i) | i in ⟨ 1 .. n ⟩ ⟩,

    aux_pascal : N1 → N1*
    aux_pascal(n) ≡
        case n of
            1 → ⟨1⟩,
            _ →
                ⟨1⟩ ⌢
                (⟨ aux_pascal(n − 1)(i − 1) + aux_pascal(n − 1)(i) | i in ⟨ 2 .. n − 1 ⟩ ⟩ ⌢ ⟨1⟩)
        end
```

3. **scheme**

```
    PAGE =
        class
            type Page = Line*, Line = Word*, Word, Dict = Word-set

            value
                is_on : Word × Page → Bool
                is_on(w, p) ≡ ( ∃ i : Nat • i ∈ inds p ∧ w ∈ elems p(i)),

                number_of : Word × Page → Nat
                number_of(w, p) ≡
                    card { (i, j) | i, j : Nat • i ∈ inds p ∧ j ∈ inds p(i) ∧ w = p(i)(j) },

                spell_check : Page × Dict → Word-set
                spell_check(p, d) ≡ { w | w : Word • is_on(w, p) ∧ w ∉ d }
        end
```

alternative definitions:

```
is_on : Word × Page → Bool
is_on(w, p) ≡ w ∈ d_elems(p),

d_elems : Page → Word-set
d_elems(p) ≡
    case p of
        ⟨⟩ → {},
        ⟨l⟩ ⌒ pr → elems l ∪ d_elems(pr)
    end,


number_of : Word × Page → Nat
number_of(w, p) ≡
    case p of
        ⟨⟩ → 0,
        ⟨l⟩ ⌒ pr → number_of(w, l) + number_of(w, pr)
    end,

number_of : Word × Line → Nat
number_of(w, l) ≡
    case l of
        ⟨⟩ → 0,
        ⟨w′⟩ ⌒ lr → if w = w′ then 1 else 0 end + number_of(w, lr)
    end
```

6 Maps

1.
```
type Report == present | not_present
value
    insert : Key × Data × Database → Database × Report
    insert(k, d, db) ≡
        if k ∈ dom db then (db † [ k ↦ d ], present)
        else (db † [ k ↦ d ], not_present)
        end,

    remove : Key × Database → Database × Report
    remove(k, db) ≡
        if k ∈ db then (db\{k}, present)
        else (db, not_present)
        end
```

2.
```
value
    merge : Database × Database → Database × Key-set
    merge(db1, db2) ≡ (db1 † db2, dom db1 ∩ dom db2)
```

3.
```
value
    report_key : (Key → Bool) × Database → Key-set
    report_key(f, db) ≡ {k | k : Key • k ∈ dom db ∧ f(k)}
```

7 Subtypes

1. (a) $\{|$ l : **Int*** • **len** l ≥ 2 $|\}$

   (b) $\{|$ l : **Int*** • **card elems** l = **len** l $|\}$

   (c) $\{|$ s : **Nat-set** • s $\neq \{\}$ $|\}$

2. $\{|$ s : T-**infset** • **card** s **post true** $|\}$
   $\{|$ l : T-**inflist** • **len** l **post true** $|\}$

3. (a) $T_2 \preceq T_1$

   (b) Are not in any subtype relation

   (c) $T_1 \preceq T_2$

   (d) $T_1 \preceq T_2$,
   $T_4 \preceq T_2$,
   $T_3 \preceq T_1$, $T_3 \preceq T_2$, $T_3 \preceq T_4$

8 Type definitions

1. **value**
   depth : Tree → **Nat**
   depth(t) ≡
   case t **of**
   nil → 0,
   node(l, v, r) → 1 + max(depth(l), depth(r))
   end

2. **value**
   is_in : **Int** × Tree → **Bool**
   is_in(i, t) ≡
   case t **of**
   nil → **false**,
   node(l, v, r) → i = v ∨ is_in(i, l) ∨ is_in(i, r)
   end

3. **type** Ordered_tree = $\{|$ t : Tree • is_ordered(t) $|\}$
   value
   is_ordered : Tree → **Bool**
   is_ordered(t) ≡
   case t **of**
   nil → **true**,
   node(l, v, r) →
   is_ordered(l) ∧
   is_ordered(r) ∧
   (∀ i : **Int** • is_in(i, l) ⇒ i < v) ∧
   (∀ i : **Int** • is_in(i, r) ⇒ i > v)
   end

4. **value**
   is_in_ordered : **Int** × Ordered_tree → **Bool**
   is_in_ordered(i, t) ≡
   case t **of**
   nil → **false**,

```
      node(l, v, r) →
        i = v ∨ i < v ∧ is_in(i, l) ∨ i > v ∧ is_in(i, r)
    end
```

5. **value**

```
  add : Int × Ordered_tree → Ordered_tree
  add(i, t) ≡
    case t of
      nil → node(nil, i, nil),
      node(l, v, r) →
        if i = v then t
        else
          if i < v then node(add(i, l), v, r)
          else node(l, v, add(i, r))
          end
        end
    end
```

6. **value**

```
  remove : Int × Ordered_tree → Ordered_tree
  remove(i, t) ≡
    case t of
      nil → nil,
      node(l, v, r) →
        if i = v then
          if l = nil then r
          else
            if r = nil then l
            else
              if depth(l) ≥ depth(r) then
                let (v1, l1) = extract_right(l) in
                  node(l1, v1, r)
                end
              else
                let (v1, r1) = extract_left(r) in
                  node(l, v1, r1)
                end
              end
            end
          end
        else
          if i < v then node(remove(i, l), v, r)
          else node(l, v, remove(i, r))
          end
        end
    end,

  extract_right : Tree ~→ (Int × Tree)
  extract_right(t) ≡
    if right(t) = nil then (val(t), left(t))
    else
      let (v1, r1) = extract_right(right(t)) in
        (v1, node(left(t), val(t), r1))
      end
    end
```

```
pre t ≠ nil,

extract_left : Tree ~→ (Int × Tree)
extract_left(t) ≡
  if left(t) = nil then (val(t), right(t))
  else
    let (v1, l1) = extract_left(left(t)) in
      (v1, node(l1, val(t), right(t)))
    end
  end
pre t ≠ nil
```

9 Imperative Specification

1. (a) x := 1; x := 2 ≡ x := 2

   (b) x := x + 1 ≡ x := x + 1

   (c) **initialise**; x ≡ **initialise**; 0

   (d) ((x := 1 ; x) ≡ (x := 0 ; x+1)) ≡ **false**

   (e) ((x := 1 ; x) = (x := 0 ; x+1)) ≡ x := 0 ; **true**

2.
```
scheme
  I_STACK1 =
    class
      type Elem

      variable st : Elem*

      value
        empty : Unit → write st Unit
        empty() ≡ st := ⟨⟩,

        push : Elem → write st Unit
        push(e) ≡ st := ⟨e⟩ ⌢ st,

        is_empty : Unit → read st Bool
        is_empty() ≡ st = ⟨⟩,

        top : Unit ~→ read st Elem
        top() ≡ hd st pre st ≠ ⟨⟩,

        pop : Unit ~→ write st Unit
        pop() ≡ st := tl st pre st ≠ ⟨⟩
    end
```

3.
```
scheme
  I_STACK2 =
    class
      type Elem

      variable st : Elem*
```

```
    value
      empty : Unit → write st Unit
      empty() post st = ⟨⟩,

      push : Elem → write st Unit
      push(e) post st = ⟨e⟩ ⁀ st`,

      is_empty : Unit → read st Bool
      is_empty() as b post b = (st = ⟨⟩),

      top : Unit ~→ read st Elem
      top() as e post e = hd st pre st ≠ ⟨⟩,

      pop : Unit ~→ write st Unit
      pop() post st = tl st` pre st ≠ ⟨⟩
  end
```

10 Concurrency

1.
```
channel l1, l2, r : Int
value
  p : Unit → in l1, l2 out r Unit
  p() ≡
    local
      variable v1, v2 : Int
    in
      (v1 := l1? ∥ v2 := l2?) ; r!max(v1,v2)
    end ;
    p()
```

2.
```
channel l1, l2, r1, r2 : Int
value
  p : Unit → in l1, l2 out r1, r2 Unit
  p() ≡
    local
      variable v1, v2 : Int
    in
      (v1 := l1? ∥ v2 := l2?) ; (r1!max(v1,v2) ∥ r2!min(v1,v2))
    end ;
    p()
```

3.
```
scheme
  SEMAPHORE =
    hide get, release, semaphore in
      class
        type Process = Unit ~→ in any out any Unit

        channel get, release : Unit

        value
          system, p1, p2, p3, f1, f2, f3, semaphore : Process
```

axiom
p1() $\equiv$ get ! () ; f1() ; release ! () ; p1(),

p2() $\equiv$ get ! () ; f2() ; release ! () ; p2(),

p3() $\equiv$ get ! () ; f3() ; release ! () ; p3(),

semaphore() $\equiv$ get? ; release? ; semaphore(),

system() $\equiv$ p1() $\parallel$ p2() $\parallel$ p3() $\parallel$ semaphore()
end

11 Implementation

1. (b) and (c) implement (a).
2. (a) - (g) have all the same maximal signature (**value** x : **Int**).

   The theories are as follows:

   (a) **axiom true**

   (b) **axiom** $x > 2$

   (c) **axiom** $x = 2$

   (d) **axiom** $x \geq 0$

   (e) **axiom** $x > 2$

   (f) **axiom** $x = 2$

   (g) **axiom false**

   Considering which theories are consequences of each other we get:

   All implement (a).
   (g) implements all.
   (b), (c), (e) and (f) implement (d).
   (b) and (e) are equivalent and therefore they implement each other.
   (c) and (f) are equivalent and therefore they implement each other.
3. (c) implements (a) and (b).
   (a) implements (b).