

# Specifying the operations performed in a Pub

Kevin Blackburn  
ICL Enterprise Engineering

March 1995

## Abstract

This note was produced a VDM-SL course presented by Peter Gorm Larsen to ICL Enterprise Engineering. The modelling of bags was one of the exercises the attendees (including myself) was confronted with during the course. Because I have a background in the Z specification language I finished this exercise before the other attendees. Thus I was asked to use the definitions of the bags to model the operations performed in a pub with a cellar and a bar. This specification is mainly intended for the purpose of illustrating how bags can be used.

## 1 Modelling of Bags

```
module BAG

exports

types
  struct Bag;
  struct Elem

functions

  Empty: () -> Bag;
  Add: Elem * Bag -> Bag;
  Remove: Elem * Bag -> Bag;
  Count: Elem * Bag -> nat;
  In: Elem * Bag -> bool;
  Join: Bag * Bag -> Bag;
  Union: Bag * Bag -> Bag;
  SubBag: Bag * Bag -> bool;
  Difference: Bag * Bag -> Bag;
  Size: Bag -> nat;
  Intersection: Bag * Bag -> Bag;
  SeqToBag: seq of Elem -> Bag

values
  бага, bagb: Bag
```

## definitions

### types

```
Elem = <A> | <B> | <C> | <D> | <E>;  
Bag = map Elem to nat1
```

### functions

```
-- Support Functions  
  
-- Minimum value of a pair of two integers  
Min : nat * nat -> nat  
Min (i, j) ==  
  if i < j  
  then i  
  else j;  
  
-- Maximum value of a pair of two integers  
Max : nat * nat -> nat  
Max (i, j) ==  
  if i > j  
  then i  
  else j;  
  
-- Add a sequence of elements, s, to a bag, b,  
-- by adding the head of s,  
-- and making a recursive call  
AuxSeqToBag : seq of Elem * Bag -> Bag  
AuxSeqToBag (s, b) ==  
  cases s :  
    []      -> b,  
    [e] ^ rest -> AuxSeqToBag(rest, Add(e, b))  
  end  
measure LenPar1;  
  
LenPar1 : seq of Elem * Bag -> nat  
LenPar1(list, -) ==  
  len list;
```

```
-- Functions Required by Customer  
-- These as described by the user document  
-- (Exercise 7)
```

```
Empty : () -> Bag  
Empty () ==  
  { |-> };
```

```
Add : Elem * Bag -> Bag
```

```

Add (e, b) ==
  if e in set dom b
  then b ++ {e |-> b(e) + 1}
  else b ++ {e |-> 1};

Remove : Elem * Bag -> Bag
Remove (e, b) ==
  if e in set dom b
  then if b(e) = 1
        then {e} <-: b
        else b ++ {e |-> b(e) - 1}
  else b;

Count : Elem * Bag -> nat
Count (e, b) ==
  if e in set dom b
  then b(e)
  else 0;
-- from given examples, if not in bag then
-- count = 0, not an error

In : Elem * Bag -> bool
In (e, b) ==
  e in set dom b;

Join : Bag * Bag -> Bag
Join (b1, b2) ==
  { e |-> Max( Count(e, b1), Count(e, b2)) |
    e in set (dom b1 union dom b2) };

Union : Bag * Bag -> Bag
Union (b1, b2) ==
  {e |-> Count (e, b1) + Count (e, b2) |
    e in set (dom b1 union dom b2)};

SubBag : Bag * Bag -> bool
SubBag (b1, b2) ==
  forall e in set dom b1 &
    Count(e, b1) <= Count(e, b2);

Difference : Bag * Bag -> Bag
Difference (b1, b2) ==
  {e |-> Count(e, b1) - Count(e, b2)
   |
   e in set dom b1
   &
   Count(e, b1) > Count(e, b2)
  };

Size : Bag -> nat

```

```

Size (b) ==
  if b = { |-> }
  then 0
  else let e in set dom b
       in
         b(e) + Size ({e} <-: b)
measure CardDom;

CardDom: Bag -> nat
CardDom(b) ==
  card dom b;

```

```

Intersection : Bag * Bag -> Bag
Intersection (b1, b2) ==
  {e |-> Min (Count(e, b1), Count(e, b2)) |
   -- Design note: Min(..., ...) is > 0
   -- as use inter in next line
   -- to ensure both Counts are at least 1
   e in set (dom b1 inter dom b2)
  };

SeqToBag : seq of Elem -> Bag
SeqToBag (s) ==
  AuxSeqToBag(s, Empty())

values
  -- The values requested by the customer for tests
  baga : Bag = { <A> |-> 3, <B> |-> 2, <C> |-> 4 };
  bagb : Bag = { <A> |-> 1, <C> |-> 5, <D> |-> 4,
                 <E> |-> 1 }

end BAG

```

## 2 Modelling of a Bar

```

module BAR
exports all
imports

from BAG

types
  Bag;
  Elem = <A> | <B> | <C> | <D> | <E>

```

### functions

```
Empty: () -> BAG`Bag;
Add: BAG`Elem * BAG`Bag -> BAG`Bag;
Remove: BAG`Elem * BAG`Bag -> BAG`Bag;
Count: BAG`Elem * BAG`Bag -> nat;
In: BAG`Elem * BAG`Bag -> bool;
Join: BAG`Bag * BAG`Bag -> BAG`Bag;
Union: BAG`Bag * BAG`Bag -> BAG`Bag;
SubBag: BAG`Bag * BAG`Bag -> bool;
Difference: BAG`Bag * BAG`Bag -> BAG`Bag;
Size: BAG`Bag -> nat;
Intersection: BAG`Bag * BAG`Bag -> BAG`Bag;
SeqToBag: seq of BAG`Elem -> BAG`Bag
```

### values

```
baga: BAG`Bag;
bagb: BAG`Bag
```

### definitions

#### types

```
Drink = BAG`Elem;
Cellar = BAG`Bag;
-- i.e. various quantities of various drinks
Bar = BAG`Bag;    -- as cellar
Supplier = seq of char;
-- Don't care about representation of suppliers
Pub = Cellar * Bar;
-- all that matters is the drink stocks in the pub
BarLevel = BAG`Bag;
-- target stocking level of bar
CellarLevel = BAG`Bag;
-- target stocking level of cellar
Stock = BAG`Bag;
Order = BAG`Bag
```

### functions

```
-- Buy an arbitrary amount of stock from
-- a supplier, assuming they have it
BuyStock : map Supplier to Stock * Supplier *
           Order * Pub -> Pub
BuyStock (supps, s, stock, mk_(c,r)) ==
  mk_(BAG`Union (c, stock), r )
pre s in set dom supps and
```

```

        BAG`SubBag( stock, supps(s));

-- Given a level of bar stocking,
-- try refilling the bar from the cellar,
-- doing the best possible
RestockBar : Pub * BarLevel -> Pub
RestockBar (mk_(c,r), bl) ==
    let missing = BAG`Difference(bl, r)
    in
    let can_restock = BAG`Intersection(missing, c)
    in
    mk_(BAG`Difference(c, can_restock),
        BAG`Union(r, can_restock));

-- A patron buys a round (list) of drinks from the bar
Round : seq of Drink * Pub -> Pub
Round (sold, mk_(c,r)) ==
    mk_(c,
        BAG`Difference(r, BAG`SeqToBag(sold))
    )
pre BAG`SubBag(BAG`SeqToBag(sold), r);

```

```

-- Given a map of suppliers and what they have,
-- work through the list of suppliers until either
-- filled requirements of cellar level or run out
-- of suppliers
RestockCellar : CellarLevel * Pub *
                map Supplier to Stock -> Pub
RestockCellar (cl, mk_(c, r), sb) ==
    if sb = { |-> }
    then mk_(c, r)
    else
        let s in set dom sb
        in
        let missing = BAG`Difference(cl, c)
        in
        if BAG`Size(missing) > 0
        then
            let can_restock = BAG`Intersection(missing, sb(s))
            in
            RestockCellar(cl,
                mk_(BAG`Union(c, can_restock), r),
                {s} <-: sb)
        else
            mk_(c, r)
measure CardCellar;

CardCellar: CellarLevel * Pub *

```

```

        map Supplier to Stock -> nat
CardCellar(-,-,sb) ==
    card dom sb;

-- Sell one drink to a patron
Drink1 : Drink * Pub -> Pub
Drink1 (dr, mk_(c,r)) ==
    mk_(c,
        BAG'Remove(dr, r))
    pre BAG'In(dr, r);

-- The pub is devoid of alcohol
Disaster : Pub -> bool
Disaster (mk_(c,r)) ==
    c = BAG'Empty() and r = BAG'Empty();

-- Return by a patron of an unopened bottle
Unwanted : Drink * Pub -> Pub
Unwanted (dr, mk_(c,r)) ==
    mk_(c,
        BAG'Add(dr, r));

-- Work out the highest single stock for
-- each kind of drink
HighestStock : map Supplier to Stock -> BAG'Bag
HighestStock (supps) ==
    if dom supps = {}
    then BAG'Empty()
    else
        let s in set dom supps
        in
            BAG'Join(supps(s), HighestStock({s} <-: supps))
measure CardDom;

CardDom: map Supplier to Stock -> nat CardDom(m) ==
    card dom m;

-- How many drinks are there in the pub
TotalDrinks : Pub -> nat
TotalDrinks (mk_(c,r)) ==
    BAG'Size(c) + BAG'Size(r)

values -- introduced for the purposes of testing
cellarlevel1 = {<A> |-> 5, <B> |-> 5, <C> |-> 3};
barlevel1 = {<A> |-> 2, <B> |-> 2, <C> |-> 5};
cellar1 = {<A> |-> 8, <B> |-> 5, <C> |-> 4};
cellar2 = {<B> |-> 1, <C> |-> 4};
bar1 = {<A> |-> 2, <B> |-> 3, <C> |-> 6};
bar2 = {<A> |-> 3, <C> |-> 2};
bar3 = {<A> |-> 3, <B> |-> 3};

```

```

pub1 = mk_(cellar1, bar1);
pub2 = mk_(cellar1, bar2);
pub3 = mk_(cellar2, bar1);
pub4 = mk_(cellar2, bar2);
pub5 = mk_(cellar1, bar3);
supps1 = {"Fizz" |-> {<A> |-> 10},
          "Real" |-> {<B> |-> 10, <C> |-> 2},
          "Scrumpy" |-> {<B> |-> 1, <C> |-> 10}};
supps2 = {"Fizz" |-> {<A> |-> 10},
          "Real" |-> {<B> |-> 1, <C> |-> 5},
          "Scrumpy" |-> {<B> |-> 1, <C> |-> 10}}

end BAR

```

### 3 Test of the BAG

```

module BAGTEST

imports from BAG all

exports all

definitions

functions

TestBagAll: () -> bool
TestBagAll() ==
  let b1 = TestAdd1(),
      b2 = TestAdd2(),
      b3 = TestCount1(),
      b4 = TestCount2(),
      b5 = TestDifference(),
      b6 = TestEmpty(),
      b7 = TestIn1(),
      b8 = TestIn2(),
      b9 = TestIntersection(),
      b10 = TestJoin(),
      b11 = TestRemove1(),
      b12 = TestRemove2(),
      b13 = TestRemove3(),
      b14 = TestSeqToBag(),
      b15 = TestSize(),
      b16 = TestSubBag1(),
      b17 = TestSubBag2(),
      b18 = TestUnion(),

```



```

in
  b1 and b2 and b3 and b4 and b5 and b6 and
  b7 and b8 and b9 and b10 and b11 and b12
  and b13 and b14 and b15 and b16 and b17
  and b18;

TestAdd1: () -> bool
TestAdd1() ==
  BAG`Add(<C>,BAG`baga) =
  { <A> |-> 3,<B> |-> 2,<C> |-> 5 };

TestAdd2: () -> bool
TestAdd2() ==
  BAG`Add(<D>,BAG`baga) =
  { <A> |-> 3,<B> |-> 2,<C> |-> 4,<D> |-> 1 };

TestCount1: () -> bool
TestCount1() ==
  BAG`Count(<D>,BAG`baga) = 0;

TestCount2: () -> bool
TestCount2() ==
  BAG`Count(<D>,BAG`bagb) = 4;

TestDifference: () -> bool
TestDifference() ==
  BAG`Difference(BAG`baga,BAG`bagb) =
  { <A> |-> 2,<B> |-> 2 };

TestEmpty: () -> bool
TestEmpty() ==
  BAG`Empty() = { |-> };

TestIn1: () -> bool
TestIn1() ==
  BAG`In(<A>,BAG`baga);

TestIn2: () -> bool
TestIn2() ==
  not BAG`In(<D>,BAG`baga);

TestIntersection: () -> bool
TestIntersection() ==
  BAG`Intersection(BAG`baga,BAG`bagb) =
  { <A> |-> 1,<C> |-> 4 };

TestJoin: () -> bool
TestJoin() ==
  BAG`Join(BAG`baga,BAG`bagb) =
  { <A> |-> 3,<B> |-> 2,<C> |-> 5,

```

```

        <D> |-> 4, <E> |-> 1 };

TestRemove1: () -> bool
TestRemove1() ==
    BAG`Remove(<A>, BAG`bagb) =
        { <C> |-> 5, <D> |-> 4, <E> |-> 1 };

TestRemove2: () -> bool
TestRemove2() ==
    BAG`Remove(<A>, BAG`baga) =
        { <A> |-> 2, <B> |-> 2, <C> |-> 4 };

TestRemove3: () -> bool
TestRemove3() ==
    BAG`Remove(<D>, BAG`baga) = BAG`baga;

TestSeqToBag: () -> bool
TestSeqToBag() ==
    BAG`SeqToBag([<A>, <A>, <B>, <C>, <A>]) =
        { <A> |-> 3, <B> |-> 1, <C> |-> 1 };

TestSize: () -> bool
TestSize() ==
    BAG`Size(BAG`baga) = 9;

TestSubBag1: () -> bool
TestSubBag1() ==
    not BAG`SubBag(BAG`baga, BAG`bagb);

TestSubBag2: () -> bool
TestSubBag2() ==
    BAG`SubBag({<A> |-> 2, <C> |-> 4}, BAG`baga);

TestUnion: () -> bool
TestUnion() ==
    BAG`Union(BAG`baga, BAG`bagb) =
        { <A> |-> 4, <B> |-> 2, <C> |-> 9,
          <D> |-> 4, <E> |-> 1 }

end BAGTEST

```