Ejemplo diseño TAD

TAD conjunto acotado

```
TAD ConjuntoAcotado {
  obs elems: conj<Nat>
  obs cota: Nat

proc conjVacio(in n: Nat): ConjuntoAcotado
   asegura res.elems = {} && res.cota= n

proc agregar(inout c: ConjuntoAcotado in e: Nat)
  requiere  e<=c.cota
  asegura c.elems = old(c).elems U {e} && c.cota=old(c).cota

proc pertenece(in c: ConjuntoAcotado, in e: Nat): Bool
   asegura res = True ⇔ e ∈ c.elems
}
```

Cota representa el máximo numero que podemos guardar

Podemos aprovechar el hecho de que los conjuntos pueden tener valores a lo sumo c.cota?

(∀n:Nat) n ∈ c.elems ➔ n<=c.cota

Representación

```
modulo ConjBitVector implementa ConjuntoAcotado{
        var arr: Array<Bool>
}
```

Aprovechamos que el conjunto esta acotado para usar la posición del array para indicar la pertenencia

```
pred InvRep(c’: ConjBitVector) {

    arr!=null && |arr| =c.cota+1
 }
```

Necesitamos espacio para todos los posibles elementos del conjunto acotado empezando de cero

```
FuncAbs(c’:ConjBitVector): ConjuntoAcotado = c |
c.cota = |arr|-1  &&
c.elems = { n:Nat | n <= c.cota && arr[n] = true) }
```

la posición n es **true** si y solo si n está en el conjunto abstracto

```
proc conjVacio(in cota: Nat): ConjuntoAcotado
   asegura res.elems = {} && res.cota= n
```

Podriamos decir
FuncAbs(res).elems = {}
FuncAbs(res).cota= n

```
proc conjVacio(in n: Int): ConjBitVector
   requiere Pre:n>=0
   asegura Pos: InvRep(res) && |res.arr|=n+1 && (∀k:Int) 0<=k<=n ➔ res[k] = False
{
   var cbv: ConjBitVector;
   cbv := new ConjBitVector;
   cbv.arr:= new Array<Bool>(n+1); //setea todo en False
   return cbv;
}
```

Para garantizar InvRep solo necesitamos |arr| = n+1
Pero para FunCAbs necesitamos todo en False

True ⇒ WpPos, código) por semántica axiomática asumida del New de Array (crea un arreglo de capacidad establecida y lo setea todo en False).

(1) asumimos que siempre hay lugar en la memoria
(2) esto puede ser caro en términos de tiempo: “O(bound)”. Mas adelante vamos a entender formalmente esto

```
proc agregar(inout c: ConjuntoAcotado in e: Nat)
  requiere  e<=c.cota
  asegura c.elems = old(c).elems U {e} && c.cota=old(c).cota
```

```
proc agregar(inout cbv: ConjBitVector, in e: Int)
  requiere Pre:InvRep(cbv) && 0<=e<=|res.arr|-1
  asegura Pos: InvRep(cbv) &&  cbv.arr[e] = True &&
    (∀n:int) 0<=n<|arr| && n≠e ➔ cbv.arr[n]=old(cbv).arr[n]
   && cbv.cota = old(cbv).cota
{
   cbv.arr[e] := True;
   return;
}
```

Notar que:
1) La Pre sale de e<=FuncAbs(cbv).cota (y |arr| =c.cota+1)
2)La Pre podría debilitarse siempre y cuando se pueda asegurar la pos

1) Por TAD FuncAbs(cbv).elems = FuncAbs (old(cbv).elems) U {e} && FuncAbs(cbv).cota=FuncAbs(old(cbv)).cota

2)La pos tiene que ser equiv o más fuerte que la pos del TAD

Nota: Pre ➔ Wp(Pos, c ódigo) x semática de la asignaciónde array (i.e setAt).
Se podría poner una precondición más debil pero habria que chequear que **e** sea más chico que la cota.

Nota: Estamos asegurando no solo corrección functional sino que InvRep se preserva

```
proc pertenece(in c: ConjuntoAcotado, in e: Nat): Bool
  asegura res = True ⇔ e ∈ c.elems
```

```
proc pertenece(in cbv: ConjBitVector, in e: Int): Bool
  requiere Pre:InvRep(cbv)
  asegura Pos: res = True ⇔  0<=e<=|arr|-1 && arr[e] = True {
  if e<= |cbv.arr|-1
     res := cbv.arr[e];
  else
    res:= False
  return res;
}
```

La pos es equiv o más fuerte que res = True ⇔ e ∈ FuncAbs(cbv).elems

TAD Conjunto

```
TAD Conjunto<T> {
   obs elems: conj<T>

proc conjVacio(): Conjunto<T>
   asegura res.elems = {}

proc agregar(inout c: Conjunto<T>, in e: T)
   asegura c.elems = old(c).elems U {e}

proc pertenece(in c: Conjunto<T>, in e: T): Bool
   asegura res = True ⇔ e ∈ c.elems
  }

proc sacar(inoutc: Conjunto<T>, in e: T)
   asegura c.elems = old(c).elems - {e}
}
```

Implementación de conjuntos

Muchas posibles implementaciones

- Arrays (redimensionando)
- Secuencias
- Listas encadenadas
- Árboles (binarios, AVL, etc)
- Tablas de Hash, etc

TAD Secuencia

```
TAD Secuencia<T> {
   obs elems: Seq<T>

proc secVacia(): Secuencia<T>
   asegura res.elems = []

proc agregar(inout s: Secuencia<T>, in pos:Int, in e: T)
   requiere 0<=pos<|elems|
   asegura s = setAt(old(s),i,e)

proc at(in s: Secuencia<T>, in pos:Int) T)
   requiere 0<=pos<|elems|
   asegura res = s.elems[pos]

proc esta(in c: Secuencia<T>, in e: T): Bool
   asegura res = True ⇔ e ∈ s.elems
  }
proc long(in c: Secuencia<T>): Nat
   asegura res = |s.elems|
  }

}
```

Implementación sobre Secuencia

```
modulo ConjSecuencia<T> implementa Conjunto<T>
{
   var sec: Secuencia<T>
}
```

```
pred InvRep(cs: ConjSecuencia<T>) {

     sinRepetidos(cs.sec.elems)
 }
```

Para simplificar la imprementación

```
aux FuncAbs(cs: ConjSecuencia<T>): Conjunto<T> = c |
mismos(c.elems,cssec.elems)
```

Hay una correspondencia uno a uno entre el conjunto y la secuencia que lo representa

```
pred sinRepetidos(s:Seq<T>){ (∀i: int ) 0 <= i < |s| ➔ apariciones(s, s[i]) = 1}

pred mismos(c:Conj<T>, s:Seq<T>){(∀e: T) apariciones(e,c) = apariciones(e,s) }
```

```
proc conjVacio(): Conjunto<T>
   asegura res.elems = {}


proc conjVacio(): ConjSecuencia<T>
   asegura res.sec.elems = []
{
    res.sec = secVacia();
    return res;
}

proc agregar(inout cs: Conjunto<T>, in e: T)
   asegura cs.elems = old(cs).elems U {e}

proc agregar(inout cs: ConjSecuencia<T>, in e: T)
   asegura e ∈ cs.sec.elems
{
   if !cs.sec.esta(e)
       cs.sec.agregar(cs.sec.long(),e)
    else
      skip;
}
```

```
proc pertenece(in c: Conjunto<T>, in e: T): Bool
   asegura res = True ⇔ e ∈ c.elems
  }

proc pertenece(in cs: ConjSecuencia<T>, in e: T): Bool
   asegura res = True ⇔ e ∈ cs.sec.elems
   var i:int
   i:= 0;
   while i<cs.sec.longitud() && cs.sec.at(i)≠e {
     i:=i+1;
   }
   res = i<cs.sec.elems.longitud();
   return res;
  }
```

e ∈ FuncAbs(cs).elems

Invariante?

```
proc sacar(inoutc: Conjunto<T>, in e: T)
   asegura c.elems = old(c).elems - {e}
}

proc sacar(inout cs: ConjSecuencia <T>, in e: T)
   asegura FuncAbs(cs).elems = FuncAbs(old(cs)).elems - {e}
{
   // Tenemos que generar una nueva secuencia sin e
   var i:int; var s:Secuencia<T>;
   i:= 0; s:= secVacia(); i

   while i<cs.sec.longitud() {
     if cs.sec.at(i)≠e then
        s.agregar(s.long(), cs.sec.elems[i])
     i:=i+1;
   }
   cs.sec = s; // reemplazo la secuencia
   return;
}
```

Podriamos decir que para todo x ≠e pasa que cs.sec.elems esta en old(cs).sec.elems, pero asi es más facil porque usamos conjuntos en vez de secuencias

Inv: apariciones(e,s) = 0 &&
0<=i<=|cs.sec.elems| &&
(∀x: T) x≠e ➔ apariciones(x,s) = apariciones(x,subseq(cs.sec.elems,0,i))