

Tema 12: Tipos abstractos de datos en PVS

José A. Alonso Jiménez

Jose-Antonio.Alonso@cs.us.es
<http://www.cs.us.es/~jalonso>

Dpto. de Ciencias de la Computación e Inteligencia Artificial

UNIVERSIDAD DE SEVILLA

Especificación de listas

- Elementos del TAD listas
 - Tipo de los elementos: T
 - Constructores: null y cons
 - Reconocedores: null? y cons?
 - Accesores: car y cdr
- Especificación del TAD lista (lista.pvs)

```
list[T: TYPE]: DATATYPE
  BEGIN
    null: null?
    cons (car: T, cdr: list): cons?
  END list
```

Generación de teorías

- Orden para generar: M-x typecheck
- Fichero generado: lista_adt.pvs
- Teorías generadas:
 - `list_adt[T: TYPE]`
 - `list_adt_map[T: TYPE, T1: TYPE]`
 - `list_adt_reduce[T: TYPE, range: TYPE]`

La teoría list_adt

- Estructura

```
list_adt[T: TYPE]: THEORY
BEGIN
...
END list_adt
```

- Signatura

```
list: TYPE
null?, cons?: [list -> boolean]
null: (null?)
cons: [[T, list] -> (cons?)]
car: [(cons?) -> T]
cdr: [(cons?) -> list]
```

La teoría list_adt

- La función ord enumera los tipos de datos del TAD (en este caso, asigna 0 a la lista vacía y 1 a las listas no vacías)

```
ord(x: list): upto(1) =  
  CASES x OF null: 0, cons(cons1_var, cons2_var): 1 ENDCASES
```

- Axiomas de extensionalidad: Dos listas con las mismas componentes son iguales

```
list_null_extensionality: AXIOM  
  FORALL (null?_var: (null?), null?_var2: (null?)):  
    null?_var = null?_var2;  
  
list_cons_extensionality: AXIOM  
  FORALL (cons?_var: (cons?), cons?_var2: (cons?)):  
    car(cons?_var) = car(cons?_var2) AND cdr(cons?_var) = cdr(cons?_var2)  
    IMPLIES cons?_var = cons?_var2;
```

La teoría list_adt

- El axioma eta: La lista construida con el primer elemento y el resto de otra lista es igual que la original

```
list_cons_eta: AXIOM
  FORALL (cons?_var: (cons?)):
    cons(car(cons?_var), cdr(cons?_var)) = cons?_var;
```

- Axioma de accesorios–constructores

- $\text{car}(\text{cons}(x, l)) = x$

```
list_car_cons: AXIOM
  FORALL (cons1_var: T, cons2_var: list):
    car(cons(cons1_var, cons2_var)) = cons1_var;
```

- $\text{cdr}(\text{cons}(x, l)) = l$

```
list_cdr_cons: AXIOM
  FORALL (cons1_var: T, cons2_var: list):
    cdr(cons(cons1_var, cons2_var)) = cons2_var;
```

La teoría list_adt

- Axiomas de partición:

- Todas las listas son simples (null?) o compuestas (cons?)

```
list_inclusive: AXIOM
  FORALL (list_var: list): null?(list_var) OR cons?(list_var);
```

- Ninguna lista es simple y compuesta

```
list_disjoint: AXIOM
  FORALL (list_var: list): NOT (null?(list_var) AND cons?(list_var));
```

La teoría list_adt

- Axioma de inducción: Sea p una propiedad sobre las listas

Si $p(\text{null})$

y $(\forall x, l)[p(l) \rightarrow p(\text{cons}(x, l))]$

entonces $(\forall l)p(l)$

```
list_induction: AXIOM
  FORALL (p: [list -> boolean]):
    (p(null) AND
      (FORALL (cons1_var: T, cons2_var: list):
        p(cons2_var) IMPLIES p(cons(cons1_var, cons2_var))))
    IMPLIES (FORALL (list_var: list): p(list_var));
```

La teoría list_adt

- El funcional every: verifica que todos los elementos de la lista cumplen la propiedad

```
every(p: PRED[T])(a: list): boolean =  
  CASES a  
    OF null: TRUE,  
       cons(cons1_var, cons2_var): p(cons1_var) AND every(p)(cons2_var)  
    ENDCASES;  
  
every(p: PRED[T], a: list): boolean =  
  CASES a  
    OF null: TRUE,  
       cons(cons1_var, cons2_var): p(cons1_var) AND every(p, cons2_var)  
    ENDCASES;
```

La teoría list_adt

- El funcional some: verifica que algún elemento de la lista cumple la propiedad

```
some(p: PRED[T])(a: list): boolean =  
  CASES a  
    OF null: FALSE,  
       cons(cons1_var, cons2_var): p(cons1_var) OR some(p)(cons2_var)  
    ENDCASES;  
  
some(p: PRED[T], a: list): boolean =  
  CASES a  
    OF null: FALSE,  
       cons(cons1_var, cons2_var): p(cons1_var) OR some(p, cons2_var)  
    ENDCASES;
```

La teoría list_adt

- La relación de sublista: $\text{subterm}(L1, L2)$ se verifica syss $L1$ es una sublista de $L2$

```
subterm(x, y: list): boolean =  
  x = y OR  
  CASES y  
    OF null: FALSE,  
       cons(cons1_var, cons2_var): subterm(x, cons2_var)  
  ENDCASES;
```

- La relación de sublista estricta: $L1 \ll L2$ se verifica si $L1$ es una sublista estricta de $L2$. La relación $\ll$ es bien fundamentada

```
<<: (well_founded?[list]) =  
  LAMBDA (x, y: list):  
    CASES y  
      OF null: FALSE,  
         cons(cons1_var, cons2_var): x = cons2_var OR x << cons2_var  
    ENDCASES;  
  
list_well_founded: AXIOM well_founded?[list](<<);
```

La teoría list_adt

- Funcional `reduce_nat` para definiciones recursivas: Sean b un número natural y $g : T \times \mathbb{N} \rightarrow \mathbb{N}$. Entonces `reduce_nat( $b, g$ )` es la función $f : \text{Listas} \rightarrow \mathbb{N}$ definida por
 - $f(L) = b$, si L es la lista vacía,
 - $f(L) = g(x, f(L'))$, si $L = \text{cons}(x, L')$

```
reduce_nat(null?_fun: nat, cons?_fun: [[T, nat] -> nat]):  
[list -> nat] =  
  LAMBDA (list_adtvar: list):  
    LET red: [list -> nat] = reduce_nat(null?_fun, cons?_fun) IN  
    CASES list_adtvar  
      OF null: null?_fun,  
        cons(cons1_var, cons2_var):  
          cons?_fun(cons1_var, red(cons2_var))  
      ENDCASES;
```

La teoría list_adt

- Funcional REDUCE_nat para definiciones recursivas: Sean $b : \text{Listas} \rightarrow \mathbb{N}$ y $g : T \times \mathbb{N} \times \text{Listas} \rightarrow \mathbb{N}$. Entonces REDUCE_nat(b, g) es la función $f : \text{Listas} \rightarrow \mathbb{N}$ definida por
 - $f(L) = b(L)$, si L es la lista vacía
 - $f(L) = g(x, f(L), L')$, si $L = \text{cons}(x, L')$

```
REDUCE_nat(null?_fun: [list -> nat],
           cons?_fun: [[T, nat, list] -> nat]):
[list -> nat] =
  LAMBDA (list_adtvar: list):
    LET red: [list -> nat] = REDUCE_nat(null?_fun, cons?_fun) IN
    CASES list_adtvar
      OF null: null?_fun(list_adtvar),
         cons(cons1_var, cons2_var):
           cons?_fun(cons1_var, red(cons2_var), list_adtvar)
    ENDCASES;
```

La teoría list_adt_map

- Estructura

```
list_adt_map[T: TYPE, T1: TYPE]: THEORY
BEGIN

  IMPORTING list_adt
  ...
END list_adt_map
```

- El funcional map: $\text{map}(f)$ es la función que para cada lista L devuelve la lista obtenida aplicando la función f a cada elemento de L

```
map(f: [T -> T1])(a: list[T]): list[T1] =
  CASES a
  OF null: null,
    cons(cons1_var, cons2_var):
      cons(f(cons1_var), map(f)(cons2_var))
  ENDCASES;
```

La teoría list_adt_reduce

- Estructura

```
list_adt_reduce[T: TYPE, range: TYPE]: THEORY
  BEGIN

    IMPORTING list_adt[T]
    ...
  END list_adt_reduce
```

La teoría list_adt_reduce

- Funcional reduce para definiciones recursivas: Sean B un conjunto, $b \in R$ y $g : T \times R \times R \rightarrow R$. Entonces $\text{reduce}(b, g)$ es la función $f : \text{Listas} \rightarrow R$ definida por
 - $f(L) = b$, si L es la lista vacía
 - $f(L) = g(x, f(L'))$, si $L = \text{cons}(x, L')$

```
reduce(null?_fun: range, cons?_fun: [[T, range] -> range]):  
[list -> range] =  
  LAMBDA (list_adtvar: list):  
    LET red: [list -> range] = reduce(null?_fun, cons?_fun) IN  
    CASES list_adtvar  
      OF null: null?_fun,  
        cons(cons1_var, cons2_var):  
          cons?_fun(cons1_var, red(cons2_var))  
      ENDCASES;
```

La teoría list_adt_reduce

- Funcional REDUCE para definiciones recursivas: Sean R un conjunto $b : \text{Listas} \rightarrow R$ y $g : T \times R \times \text{Listas} \rightarrow R$. Entonces $\text{REDUCE}(b, g)$ es la función $f : \text{Listas} \rightarrow R$ definida por
 - $f(L) = b(L)$, si L es la lista vacía
 - $f(L) = g(x, f(L'), \text{cons}(x, L'))$

```
REDUCE(null?_fun: [list -> range],
      cons?_fun: [[T, range, list] -> range]):
[list -> range] =
  LAMBDA (list_adtvar: list):
    LET red: [list -> range] = REDUCE(null?_fun, cons?_fun) IN
    CASES list_adtvar
      OF null: null?_fun(list_adtvar),
         cons(cons1_var, cons2_var):
           cons?_fun(cons1_var, red(cons2_var), list_adtvar)
    ENDCASES;
```

Propiedades de listas

- Estructura de la teoría lista_props

```
lista_props[T: TYPE]: THEORY
BEGIN
  % IMPORTING list_adt[T]
  % Comentario: No es necesario porque la teoría list es del preludio

  l, l1, l2, l3, ac: VAR list[T]
  x: VAR T
  P: VAR pred[T]
  ...
END lista_props
```

- Definición de la longitud

```
longitud(l): RECURSIVE nat =
  CASES 1 OF
    null: 0,
    cons(x, l1): longitud(l1) + 1
  ENDCASES
  MEASURE 1 BY <<
```

Propiedades de listas

- TCC generado por la definición de longitud

```
% Termination TCC generated (at line 18, column 19) for longitud(l1)
% proved - complete
longitud_TCC1: OBLIGATION
  FORALL (l1: list[T], x: T, l: list[T]): l = cons(x, l1) IMPLIES l1 << l;
```

- Definicion de la relacion de pertenencia

```
pertenece(x, l): RECURSIVE bool =
  CASES 1 OF
    null: FALSE,
    cons(y, l1): x = y OR pertenece(x, l1)
  ENDCASES
  MEASURE 1 BY <<
```

Propiedades de listas

- TCC generado

```
% Termination TCC generated (at line 38, column 28) for pertenece(x, l1)
% proved - complete
pertenece_TCC1: OBLIGATION
  FORALL (l1: list[T], y: T, l: list[T], x: T):
    l = cons(y, l1) AND NOT x = y IMPLIES l1 << l;
```

- Lema: $x \in L \rightarrow L \neq \emptyset$

```
pertenece_vacia: LEMMA
  pertenece(x, l) IMPLIES NOT null?(l)
```

- Prueba con (grind)

Propiedades de listas

- Concatenacion de listas

```
concatenacion(l1, l2): RECURSIVE list[T] =  
  CASES l1 OF  
    null: l2,  
    cons(x, l): cons(x, concatenacion(l, l2))  
  ENDCASES  
  MEASURE l1 BY <<
```

- TCC generado

```
% Termination TCC generated (at line 57, column 26) for  
  % concatenacion(l, l2)  
  % proved - complete  
concatenacion_TCC1: OBLIGATION  
  FORALL (l: list[T], x: T, l1: list[T]): l1 = cons(x, l) IMPLIES l << l1;
```

Propiedades de listas

- Lema: Al añadir la lista vacía a una lista L se obtiene L

```
concatenacion_con_nulo: LEMMA  
  concatenacion(l, null) = l
```

- Prueba con (induct-and-simplify "l")

- Lema

```
concatenacion_lista_unitaria: LEMMA  
  concatenacion(cons(x,null),l) = cons(x,l)
```

- Prueba con (grind)

- Lema

```
asociatividad_concatenacion: LEMMA  
  concatenacion(concatenacion(l1,l2),l3) =  
  concatenacion(l1,concatenacion(l2,l3))
```

- Prueba con (induct-and-simplify "l1")

Propiedades de listas

- Definicion de inversa de una lista

```
inversa(l): RECURSIVE list[T] =  
  CASES l OF  
    null: l,  
    cons(x,l1): concatenacion(inversa(l1),cons(x,null))  
  ENDCASES  
  MEASURE l BY <<
```

- Lema

```
inversa_de_concatenacion: LEMMA  
  inversa(concatenacion(l1,l2)) =  
  concatenacion(inversa(l2),inversa(l1))
```

- Prueba

inversa_de_concatenacion :

```
|-----  
{1}  FORALL (l1, l2: list[T]):  
      inversa(concatenacion(l1, l2)) =  
      concatenacion(inversa(l2), inversa(l1))
```

Propiedades de listas

```
Rule? (induct-and-simplify "l1")
concatenacion rewrites concatenacion(null, l2!1)
  to l2!1
inversa rewrites inversa(null)
  to null
concatenacion rewrites concatenacion(cons(cons1_var!1, cons2_var!1), l2!1)
  to cons(cons1_var!1, concatenacion(cons2_var!1, l2!1))
inversa rewrites inversa(cons(cons1_var!1, concatenacion(cons2_var!1, l2!1)))
  to concatenacion(inversa(concatenacion(cons2_var!1, l2!1)),
    cons(cons1_var!1, null))
inversa rewrites inversa(cons(cons1_var!1, cons2_var!1))
  to concatenacion(inversa(cons2_var!1), cons(cons1_var!1, null))
By induction on l1, and by repeatedly rewriting and simplifying,
this yields 2 subgoals:
```

Propiedades de listas

`inversa_de_concatenacion.1 :`

```
|-----  
{1}  inversa(l2!1) = concatenacion(inversa(l2!1), null)
```

Rule? (rewrite "concatenacion_con_nulo")

Found matching substitution:

`l: list[T] gets inversa(l2!1),`

Rewriting using `concatenacion_con_nulo`, matching in `*`,

This completes the proof of `inversa_de_concatenacion.1`.

Propiedades de listas

`inversa_de_concatenacion.2 :`

```
{-1}  inversa(concatenacion(cons2_var!1, l2!1)) =  
      concatenacion(inversa(l2!1), inversa(cons2_var!1))  
  |-----  
{1}  concatenacion(inversa(concatenacion(cons2_var!1, l2!1)),  
                  cons(cons1_var!1, null))  
      =  
      concatenacion(inversa(l2!1),  
                  concatenacion(inversa(cons2_var!1),  
                              cons(cons1_var!1, null)))
```

Propiedades de listas

Rule? (replace -1)
Replacing using formula -1,
this simplifies to:
inversa_de_concatenacion.2 :

```
[-1]  inversa(concatenacion(cons2_var!1, l2!1)) =  
      concatenacion(inversa(l2!1), inversa(cons2_var!1))  
  |-----  
{1}  concatenacion(concatenacion(inversa(l2!1), inversa(cons2_var!1)),  
                  cons(cons1_var!1, null))  
      =  
      concatenacion(inversa(l2!1),  
                  concatenacion(inversa(cons2_var!1),  
                              cons(cons1_var!1, null)))
```

Propiedades de listas

Rule? (rewrite "asociatividad_concatenacion")

Found matching substitution:

l3: list[T] gets cons(cons1_var!1, null),

l2 gets inversa(cons2_var!1),

l1 gets inversa(l2!1),

Rewriting using asociatividad_concatenacion, matching in *,

This completes the proof of inversa_de_concatenacion.2.

Q.E.D.

Propiedades de listas

- Lema

```
inversa_unitaria: LEMMA  
  inversa(cons(x,null)) = cons(x,null)
```

- Prueba con (grind)

- Teorema

```
inversa_inversa: THEOREM  
  inversa(inversa(l)) = l
```

- Prueba con (induct-and-simplify "l" :theories "lista_props")

Especificación de árboles binarios

- Elementos del TAD árbol binario
 - Tipo de los nodos: T
 - Constructores: hoja y nodo
 - Reconocedores: hoja? y nodo?
 - Accesos: val, izquierda y derecha
- Especificación del TAD árbol binario (arbol_binario.pvs)

```
arbol_binario[T: TYPE]: DATATYPE
BEGIN
  hoja: hoja?
  nodo(val: T, izquierda: arbol_binario, derecha: arbol_binario): nodo?
END arbol_binario
```

Generación de teorías

- Orden para generar: M-x typecheck
- Fichero generado: `arbol_binario_adt.pvs`
- Teorías generadas:
 - `arbol_binario_adt[T: TYPE]`
 - `arbol_binario_adt_map[T: TYPE, T1: TYPE]`
 - `arbol_binario_adt_reduce[T: TYPE, range: TYPE]`

La teoría `arbol_binario_adt`

- Estructura

```
arbol_binario_adt[T: TYPE]: THEORY
BEGIN
  ...
END arbol_binario_adt
```

- Signatura

```
arbol_binario: TYPE
hoja?, nodo?: [arbol_binario -> boolean]
hoja: (hoja?)
nodo: [[T, arbol_binario, arbol_binario] -> (nodo?)]
val: [(nodo?) -> T]
izquierda: [(nodo?) -> arbol_binario]
derecha: [(nodo?) -> arbol_binario]
```

La teoría arbol_binario_adt

- La función ord enumera los tipos de datos del TAD (en este caso, asigna 0 a las hojas y 1 a los nodos)

```
ord(x: arbol_binario): upto(1) =  
  CASES x OF hoja: 0, nodo(nodo1_var, nodo2_var, nodo3_var): 1 ENDCASES
```

- Axiomas de extensionalidad: Dos árboles con las mismas componentes son iguales

```
arbol_binario_hoja_extensionality: AXIOM  
  FORALL (hoja?_var: (hoja?), hoja?_var2: (hoja?)):  
    hoja?_var = hoja?_var2;  
  
arbol_binario_nodo_extensionality: AXIOM  
  FORALL (nodo?_var: (nodo?), nodo?_var2: (nodo?)):  
    val(nodo?_var) = val(nodo?_var2) AND  
    izquierda(nodo?_var) = izquierda(nodo?_var2) AND  
    derecha(nodo?_var) = derecha(nodo?_var2)  
    IMPLIES nodo?_var = nodo?_var2;
```

La teoría arbol_binario_adt

- El axioma eta: El árbol construido con el valor, la rama izquierda y la rama derecha de otro árbol es idéntico al original

```
arbol_binario_nodo_eta: AXIOM
  FORALL (nodo?_var: (nodo?)):
    nodo(val(nodo?_var), izquierda(nodo?_var), derecha(nodo?_var)) =
      nodo?_var;
```

- Axioma de accesorios—constructores

- $\text{val}(\text{nodo}(v, A, B)) = v$

```
arbol_binario_val_nodo: AXIOM
  FORALL (nodo1_var: T, nodo2_var: arbol_binario,
          nodo3_var: arbol_binario):
    val(nodo(nodo1_var, nodo2_var, nodo3_var)) = nodo1_var;
```

La teoría arbol_binario_adt

- $\text{izquierda}(\text{nodo}(v, A, B)) = A$

```
arbol_binario_izquierda_nodo: AXIOM
  FORALL (nodo1_var: T, nodo2_var: arbol_binario, nodo3_var: arbol_binario):
    izquierda(nodo(nodo1_var, nodo2_var, nodo3_var)) = nodo2_var;
```

- $\text{derecha}(\text{nodo}(v, A, B)) = B$

```
arbol_binario_derecha_nodo: AXIOM
  FORALL (nodo1_var: T, nodo2_var: arbol_binario, nodo3_var: arbol_binario):
    derecha(nodo(nodo1_var, nodo2_var, nodo3_var)) = nodo3_var;
```

- **Axiomas de partición:** Todos los árboles binarios son simples (hoja?) o compuestos (nodo?) de manera excluyente

```
arbol_binario_inclusive: AXIOM
  FORALL (arbol_binario_var: arbol_binario):
    hoja?(arbol_binario_var) OR nodo?(arbol_binario_var);
arbol_binario_disjoint: AXIOM
  FORALL (arbol_binario_var: arbol_binario):
    NOT (hoja?(arbol_binario_var) AND nodo?(arbol_binario_var));
```

La teoría `arbol_binario_adt`

- Axioma de inducción: Sea p una propiedad sobre los árboles.

Si $p(\text{hoja})$

y $(\forall v, A_1, A_2)[p(A_1) \wedge p(A_2) \rightarrow p(\text{nodo}(v, A_1, A_2))]$,

entonces $(\forall A)p(A)$

```
arbol_binario_induction: AXIOM
  FORALL (p: [arbol_binario -> boolean]):
    (p(hoja) AND
      (FORALL (nodo1_var: T, nodo2_var: arbol_binario,
                nodo3_var: arbol_binario):
        p(nodo2_var) AND p(nodo3_var) IMPLIES
          p(nodo(nodo1_var, nodo2_var, nodo3_var))))
    IMPLIES
      (FORALL (arbol_binario_var: arbol_binario): p(arbol_binario_var));
```

La teoría arbol_binario_adt

- El funcional every: verifica que todos los valores de los nodos del árbol cumple la propiedad

```
every(p: PRED[T])(a: arbol_binario): boolean =  
  CASES a  
    OF hoja: TRUE,  
       nodo(nodo1_var, nodo2_var, nodo3_var):  
         p(nodo1_var) AND every(p)(nodo2_var) AND every(p)(nodo3_var)  
    ENDCASES;  
  
every(p: PRED[T], a: arbol_binario): boolean =  
  CASES a  
    OF hoja: TRUE,  
       nodo(nodo1_var, nodo2_var, nodo3_var):  
         p(nodo1_var) AND every(p, nodo2_var) AND every(p, nodo3_var)  
    ENDCASES;
```

La teoría arbol_binario_adt

- El funcional `some`: verifica que alguno de los valores de los nodos del árbol cumple la propiedad

```
some(p: PRED[T])(a: arbol_binario): boolean =  
  CASES a  
    OF hoja: FALSE,  
       nodo(nodo1_var, nodo2_var, nodo3_var):  
         p(nodo1_var) OR some(p)(nodo2_var) OR some(p)(nodo3_var)  
    ENDCASES;  
  
some(p: PRED[T], a: arbol_binario): boolean =  
  CASES a  
    OF hoja: FALSE,  
       nodo(nodo1_var, nodo2_var, nodo3_var):  
         p(nodo1_var) OR some(p, nodo2_var) OR some(p, nodo3_var)  
    ENDCASES;
```

La teoría arbol_binario_adt

- La relación de subárbol: $\text{subterm}(A,B)$ se verifica syss A es un subárbol de B

```
subterm(x, y: arbol_binario): boolean =  
  x = y OR  
  CASES y  
    OF hoja: FALSE,  
       nodo(nodo1_var, nodo2_var, nodo3_var):  
         subterm(x, nodo2_var) OR subterm(x, nodo3_var)  
    ENDCASES;
```

La teoría arbol_binario_adt

- La relación de subárbol estricto: $A \ll B$ se verifica si A es un subárbol estricto de B. La relación $\ll$ es bien fundamentada

```
<<: (well_founded?[arbol_binario]) =  
  LAMBDA (x, y: arbol_binario):  
    CASES y  
      OF hoja: FALSE,  
         nodo(nodo1_var, nodo2_var, nodo3_var):  
           (x = nodo2_var OR x << nodo2_var) OR  
           x = nodo3_var OR x << nodo3_var  
      ENDCASES;  
  
arbol_binario_well_founded: AXIOM well_founded?[arbol_binario](<<);
```

La teoría `arbol_binario_adt`

- Funcional `reduce_nat` para definiciones recursivas: Sean b un número natural y $g : T \times \mathbb{N} \rightarrow \mathbb{N}$. Entonces `reduce_nat( $b, g$ )` es la función $f : \text{Arboles} \rightarrow \mathbb{N}$ definida por
 - $f(A) = b$, si A es una hoja
 - $f(A) = g(v, f(A_1), f(A_2))$, si $A = \text{nodo}(v, A_1, A_2)$

```
reduce_nat(hoja?_fun: nat, nodo?_fun: [[T, nat, nat] -> nat]):  
[arbol_binario -> nat] =  
  LAMBDA (arbol_binario_adtvar: arbol_binario):  
    LET red: [arbol_binario -> nat] = reduce_nat(hoja?_fun, nodo?_fun)  
    IN  
    CASES arbol_binario_adtvar  
      OF hoja: hoja?_fun,  
         nodo(nodo1_var, nodo2_var, nodo3_var):  
           nodo?_fun(nodo1_var, red(nodo2_var), red(nodo3_var))  
      ENDCASES;
```

La teoría `arbol_binario_adt`

- Funcional `REDUCE_nat` para definiciones recursivas: Sean $b : \text{Arboles} \rightarrow \mathbb{N}$ y $g : T \times \mathbb{N} \times \text{Arboles} \rightarrow \mathbb{N}$. Entonces $\text{REDUCE_nat}(b, g)$ es la función $f : \text{Arboles} \rightarrow \mathbb{N}$ definida por
 - $f(A) = b(A)$, si A es una hoja
 - $f(A) = g(v, f(A_1), f(A_2), A)$, si $A = \text{nodo}(v, A_1, A_2)$

```
REDUCE_nat(hoja?_fun: [arbol_binario -> nat],
           nodo?_fun: [[T, nat, nat, arbol_binario] -> nat]):
[arbol_binario -> nat] =
  LAMBDA (arbol_binario_adtvar: arbol_binario):
    LET red: [arbol_binario -> nat] = REDUCE_nat(hoja?_fun, nodo?_fun) IN
      CASES arbol_binario_adtvar
        OF hoja: hoja?_fun(arbol_binario_adtvar),
           nodo(nodo1_var, nodo2_var, nodo3_var):
             nodo?_fun(nodo1_var, red(nodo2_var), red(nodo3_var),
                       arbol_binario_adtvar)
        ENDCASES;
```

La teoría `arbol_binario_adt_map`

- Estructura

```
arbol_binario_adt_map[T: TYPE, T1: TYPE]: THEORY
BEGIN

  IMPORTING arbol_binario_adt
  ...
END arbol_binario_adt_map
```

- El funcional `map`: `map(f)` es la función que para cada árbol `A` devuelve el árbol obtenido aplicando la función `f` al valor de cada nodo de `A`

```
map(f: [T -> T1])(a: arbol_binario[T]): arbol_binario[T1] =
  CASES a
  OF hoja: hoja,
    nodo(nodo1_var, nodo2_var, nodo3_var):
      nodo(f(nodo1_var), map(f)(nodo2_var), map(f)(nodo3_var))
  ENDCASES;
```

La teoría `arbol_binario_adt_reduce`

- Estructura

```
arbol_binario_adt_reduce[T: TYPE, range: TYPE]: THEORY
  BEGIN

    IMPORTING arbol_binario_adt[T]
    ...
  END arbol_binario_adt_reduce
```

La teoría `arbol_binario_adt_reduce`

- Funcional `reduce` para definiciones recursivas: Sean B un conjunto, $b \in R$ y $g : T \times R \rightarrow R$. Entonces `reduce( $b, g$ )` es la función $f : \text{Arboles} \rightarrow R$ definida por
 - $f(A) = b$, si A es una hoja
 - $f(A) = g(v, f(A_1), f(A_2))$, si $A = \text{nodo}(v, A_1, A_2)$

```
reduce(hoja?_fun: range, nodo?_fun: [[T, range, range] -> range]):  
[arbol_binario -> range] =  
  LAMBDA (arbol_binario_adtvar: arbol_binario):  
    LET red: [arbol_binario -> range] = reduce(hoja?_fun, nodo?_fun) IN  
      CASES arbol_binario_adtvar  
        OF hoja: hoja?_fun,  
           nodo(nodo1_var, nodo2_var, nodo3_var):  
             nodo?_fun(nodo1_var, red(nodo2_var), red(nodo3_var))  
        ENDCASES;
```

La teoría `arbol_binario_adt_reduce`

- Funcional REDUCE para definiciones recursivas: Sean R un conjunto $b : \text{Arboles} \rightarrow R$ y $g : T \times R \times \text{Arboles} \rightarrow R$. Entonces $\text{REDUCE}(b, g)$ es la función $f : \text{Arboles} \rightarrow R$ definida por
 - $f(A) = b(A)$, si A es una hoja
 - $f(A) = g(v, f(A_1), f(A_2), A)$, si $A = \text{nodo}(v, A_1, A_2)$

```
REDUCE(hoja?_fun: [arbol_binario -> range],
      nodo?_fun: [[T, range, range, arbol_binario] -> range]):
[arbol_binario -> range] =
  LAMBDA (arbol_binario_adtvar: arbol_binario):
    LET red: [arbol_binario -> range] = REDUCE(hoja?_fun, nodo?_fun) IN
      CASES arbol_binario_adtvar
        OF hoja: hoja?_fun(arbol_binario_adtvar),
           nodo(nodo1_var, nodo2_var, nodo3_var):
             nodo?_fun(nodo1_var, red(nodo2_var), red(nodo3_var),
                       arbol_binario_adtvar)
        ENDCASES;
```

Arboles binarios sobre los naturales

- Estructura

```
arbol_binario_props: THEORY
  BEGIN
    IMPORTING arbol_binario_adt[nat]
    ...
  END arbol_binario_props
```

- Ejemplos

```
A1: arbol_binario = hoja
A2: arbol_binario = nodo(7,A1,A1)
A3: arbol_binario = nodo(5,A1,A2)
A4: arbol_binario = nodo(6,A2,A3)
```

- Propiedades (se prueban con (grind))

```
L1: LEMMA nodo?(A4)
L2: LEMMA every(odd?)(A3)
L3: LEMMA some(even?)(A4)
```

Arboles binarios sobre los naturales

- La suma(A) es la suma de los valores de los nodos de A

```
suma(A: arbol_binario): RECURSIVE nat =  
  CASES A OF  
    hoja: 0,  
    nodo(v,A1,A2): v+suma(A1)+suma(A2)  
  ENDCASES  
  MEASURE A BY <<
```

- Condiciones generadas (y probadas)

```
suma_TCC1: OBLIGATION  
  FORALL (A1, A2: arbol_binario[nat], v: nat, A: arbol_binario):  
    A = nodo(v, A1, A2) IMPLIES A1 << A;  
  
suma_TCC2: OBLIGATION  
  FORALL (A1, A2: arbol_binario[nat], v: nat, A: arbol_binario):  
    A = nodo(v, A1, A2) IMPLIES A2 << A;
```

Arboles binarios sobre los naturales

- Cálculo mediante prueba

```
L4: LEMMA suma(A4) = 25
```

- Se prueba con (grind)

- Cálculo mediante M-x pvs-ground-evaluator

```
<GndEval> "suma(A4)"  
==>  
25
```