Capítulo 1

Satisfabilidad de expresiones booleanas

1.1 Enunciado

Decimos que una expresión o fórmula booleana sobre variables, con los operadores $\wedge$, $\vee$ y $\neg$, es **satisfactible** si existe una forma de sustituir sus variables por valores que permitan evaluarla a *cierto*.

Ejemplo: la expresión

$$(x_1 \vee x_2) \wedge (\neg x_2 \vee x_3) \wedge \neg x_1$$

es satisfactible pues si x_1 es *falso*, x_2 es *cierto* y x_3 es *cierto* la expresión se evalúa a *cierto*.

En cambio, la expresión

$$(x_1 \vee \neg x_3) \wedge (\neg x_2 \vee x_3) \wedge (x_2 \vee x_1) \wedge \neg x_1$$

es insatisfactible.

Una manera de detectar si una expresión booleana es satisfactible consiste en comprobar exhaustivamente todos las posibles sustituciones de valores para sus variables. Si en algún caso se evalúa a *cierto* entonces es satisfactible.

SE PIDE: un **programa modular** que, dada una expresión booleana codificada en una secuencia de enteros, determine si tal expresión es satisfactible mediante un proceso iterativo basado en la estrategia anterior. En caso de que la expresión sea satisfactible, el programa ha de devolver unos valores de sus variables para los que se evalúe a *cierto*. Las funciones que realicen recorridos o búsquedas sobre vectores han de especificarse y derivarse formalmente.

La codificación de la expresión en la entrada del programa será la siguiente:

- la expresión aparece en notación prefija, es decir, la fórmula

  $$(x_1 \vee x_2) \wedge (\neg x_2 \vee x_3) \wedge \neg x_1$$

  se representa por

  $$\wedge(\wedge(\vee(x_1, x_2), \vee(\neg(x_2), x_3)), \neg(x_1))$$

- las variables $x_1, x_2, \ldots, x_n$ se codifican con los valores $1, 2, \cdots, n$, mientras que las operaciones $\wedge$, $\vee$ y $\neg$ se codifican con $-1, -2$ y -3 respectivamente. Dado que las aridades de los operadores son conocidas podemos prescindir de los paréntesis y las comas.

Así, la secuencia que codifica la expresión

$$\wedge(\wedge(\vee(x_1, x_2), \vee(\neg(x_2), x_3)), \neg(x_1))$$

es

$$-1\ -1\ -2\ 1\ 2\ -2\ -3\ 2\ 3\ -3\ 1$$

Como suponemos que la secuencia siempre contiene una expresión correcta, no es necesario que la secuencia tenga marca.

El número máximo de variables distintas que puede contener una expresión está acotado por la constante MAX_VAR (p.e. 10). Las variables que aparecen en la expresión están numeradas correlativamente.

1.2 Esquema preliminar del programa principal

Para especificar el problema usaremos una notación no totalmente formal pero con la suficiente precisión:

S : secuencia de enteros;
sat : booleano;
{S contiene una expresión booleana sintácticamente correcta }
  Satisfabilidad
{*sat* = la expresión contenida en S es satisfactible y, en caso afirmativo, obtenemos una solución}

Del enunciado se desprende que nuestro problema consiste en una búsqueda sobre el conjunto de posibles soluciones para la expresión dato, una vez que ésta haya sido leída y almacenada.

Proponemos el esquema

```
leer expresión
obtener candidato
probar candidato
{Inv: sat = alguno de los candidatos probados satisface la expresión
 y, en caso afirmativo, el último probado es una solución}
{Cota: número de candidatos que quedan por probar }
mientras no sat y quedan candidatos hacer
    obtener candidato
    probar candidato
fmientras
escribir resultados
```

Claramente, aparecen dos tipos de datos que dan lugar a sendos módulos. En el módulo `Expresión` deberemos incluir el tipo correspondiente más las operaciones asociadas.

Estas consistirán en crear una expresión que consta de una sola variable y, por otra parte, conjuntar, disyuntar y negar expresiones. Por último, necesitaremos saber si un candidato satisface una expresión y el número de variables de una expresión. Si la expresión de la entrada tiene sus variables numeradas correlativamente a partir de 1, ese valor coincide con el máximo de los índices de las variables, y ésta será la función que incluiremos.

En el módulo `Candidato` habrá que incluir el tipo del mismo nombre y operaciones que permitan obtener de manera ordenada, según algún criterio que discutiremos en su momento, todos los candidatos posibles para una expresión con el fin de poder probarlos. También necesitaremos consultar qué valor le asigna un candidato a cada variable de la expresión.

1.3 Especificación del módulo `Expresión`

En nuestro problema una expresión puede ser una variable, o bien el resultado de conjuntar, disyuntar o negar otras expresiones. Mantendremos el criterio de que las variables van numeradas desde 1 hasta un cierto MAX_VAR. De esta forma, podemos crear una variable simplemente indicando cuál es su índice.

```
Módulo Expresión;

{ Tipo de módulo: datos }

Especificación;

Const MAX_VAR=10;

Tipo expresión;

{Descripción general: representa las expresiones booleanas con <=MAX_VAR
variables que se pueden construir mediante las operaciones de conjunción,
disyunción y negación}

Operaciones

función crea_expresión(i :nat) ret e: expresión;
{Pre: 1 =< i =< MAX_VAR }
{Post: e es una expresión formada solamente por la variable x_i}

función conjunta_expresión(e1, e2: expresión) ret e: expresión;
{Pre: cierto}
{Post: e = e1 /\ e2}
```

```
función disyunta_expresión(e1, e2: expresión) ret e: expresión;
{Pre: cierto}
{Post: e = e1 \/ e2}

función niega_expresión(e1: expresión) ret e: expresión;
{Pre: cierto}
{Post: e = !e1}

función consulta_max_var(e: expresión) ret n: natural;
{Pre: cierto}
{Post: n = índice máximo de las variables de e}

función satisface_expresión(e: expresión; c: candidato) ret b: booleano;
{Pre: c contiene al menos tantos valores como el índice máximo de las
variables de e}
{Post: b = c satisface e}

FEspecificación
```

1.4 Especificación del módulo `Candidato`

Una expresión booleana con n variables tiene 2^n candidatos a solución, ya que cada variable tiene dos posibles valores (*cierto* y *falso*). Dado que vamos a probar diferentes candidatos hasta encontrar uno que la satisfaga o hasta agotarlos todos, dotaremos al tipo `candidato` de un orden y de operaciones tales como `primer_candidato`, `siguiente_candidato` y `último_candidato`, respecto a ese orden, con las cuales podremos construir todos los candidatos. Por otra parte, necesitamos una operación `consulta_valor` que indique qué valor le proporciona un candidato a una variable dada.

```
Modulo Candidato;

{ Tipo de módulo: datos }

Especificación;

Const MAX_VAR = 10;

Tipo candidato;

{Descripción general: representa un candidato a solución para una
expresión booleana con hasta MAX_VAR variables, es decir, un conjunto de
hasta MAX_VAR valores booleanos}
```

```
Operaciones

función primer_candidato (num_var: nat) ret c:candidato;
{Pre: 1 =< num_var =< MAX_VAR }
{Post: c es el primer candidato para una expresión de num_var variables }

función siguiente_candidato (c1:candidato) ret c2:candidato;
{Pre: c1 no es el último candidato para una expresión del número
correspondiente de variables}
{Post: c2 es el siguiente candidato a c1 }

función último_candidato (c:candidato) ret b:booleano;
{Pre: }
{Post: b = c es el último candidato para una expresión del número
correspondiente de variables }

función consulta_valor (c:candidato; i:natural) ret b:booleano;
{Pre: 1 <= i <= número de valores de c}
{Post: b =  valor que c asocia a la i-ésima variable}

FEspecificación
```

1.5 Programa principal detallado

Ahora ya podemos escribir el programa principal con todo detalle, pues disponemos de la especificación de todos los elementos pertenecientes a los otros módulos.

```
Programa Principal;

Usa Expresión,Candidato;

Const MAX_VAR = 10;

Var e:expresión; n:nat; c:candidato;

{Pre: la secuencia de entrada contiene una expresión booleana válida con
MAX_VAR variables como máximo y todas ellas numeradas correlativamente
empezando por 1}

e:=leer_expresión();
n:=consulta_max_var(e);
```

```
c:=primer_candidato(n);
sat:=satisface_expresión(e,c);
{Inv: sat = alguno de los candidatos probados satisface la expresión
y en caso afirmativo c es una solución}
{Cota: número de candidatos que quedan por probar}
 mientras !(sat) y !(último_candidato(c)) hacer
   sat:=satisface_expresión(e,siguiente(c));
   c:=siguiente(c);
 fmientras;

{Post: sat = la expresión contenida en la secuencia de entrada es
satisfactible y en caso afirmativo c es una solución}

si !sat entonces escribir(``la expresión no es satisfactible'')
[] sat entonces escribir(``la expresión es satisfactible'');
                escribir(`` y se satisface con '', c)
fsi
```

Falta por diseñar la operación de lectura de una expresión. Podemos aprovechar que estas tienen una estructura eminentemente recursiva, plasmando ésta en el algoritmo:

```
función leer_expresión() ret e:expresión
{Pre: la secuencia de entrada contiene una expresión booleana válida}

 var n: entero; e1, e2: expresión;

 n:=leer();
 si c>0 entonces e:=crea_expresión(c);
 [] c=-1 entonces e1:=leer_expresión();
                  e2:=leer_expresión();
                  e:=conjunta_expresión(e1,e2);
 [] c=-2 entonces e1:=leer_expresión();
                  e2:=leer_expresión();
                  e:=disyunta_expresión(e1,e2);
 [] c=-3 entonces e1:=leer_expresión();
                  e:=niega_expresión(e1);
 fsi

{Post: e es la expresión contenida en la secuencia de entrada}
```

1.6 Implementación del módulo `Expresión`

En esta sección supondremos que nos han pedido justificar **informalmente** el diseño de las operaciones del tipo `Expresión`.

Dado que los operadores booleanos son binarios (conjunción y disyunción) o unarios (negación), parece natural representar una expresión mediante un árbol binario de enteros, en el que los nodos internos son los operadores y las hojas son las variables. Usamos la misma codificación que en la entrada: positivos para las variables y -1, -2 y -3 para los operadores. Para saber cuántas variables contiene la expresión guardaremos también un natural, con lo que necesitamos una tupla.

```
Módulo Expresión;

{ Tipo de módulo: datos }

Implementación;

Tipo expresión =
   tupla
     arb_exp : árbol de enteros
     var_max : natural
   ftupla;

{El campo arb_exp representa la expresión en sí, guardada en un árbol, y
var_max es el máximo índice de sus variables, de modo que si éstas están
numeradas correlativamente a partir de 1, equivale al número de variables
```

Pasamos ahora a implementar las operaciones justificando al mismo tiempo nuestras decisiones de diseño.

1.6.1 Implementación de `crea_expresión`

Justificación: La operación `crea_expresión` produce una expresión formada por una única variable. Por lo tanto, el árbol de dicha expresión tendrá un único nodo, formado por el índice de dicha variable, que además será el máximo.

Código:

```
función crea_expresión(i: nat) ret e: expresión;
{Pre: 1 =< i =< MAX }

   e.arb_exp := plantar(i,a_n,a_n);
   e.var_max := i

{Post: e es una expresión formada solamente por la variable x_i}
```

1.6.2 Implementación de `conjunta_expresión`

Justificación: La operación `conjunta_expresión` produce el resultado de conjuntar dos expresiones. El arbol de dicha expresión sera el formado al plantar el operador -1 sobre los árboles de las expresiones dato y el máximo de los índices será el máximo de las dos.

Código:

```
función conjunta_expresión(e1, e2: expresión) ret e: expresión;
{Pre: cierto}

   e.arb_exp := plantar (-1, e1.arb_exp, e2.arb_exp);
   e.var_max := max(e1.var_max, e2.var_max)

{Post: e = e1 /\ e2}
```

1.6.3 Implementación de `disyunta_expresión`

Justificación: La operación `disyunta_expresión` produce el resultado de conjuntar dos expresiones. La construcción de dicha expresión es similar a la de la función anterior, pero plantando esta vez el operador -2.

Código:

```
función disyunta_expresión(e1, e2: expresión) ret e: expresión;
{Pre: cierto}

   e.arb_exp := plantar (-2, e1.arb_exp, e2.arb_exp);
   e.var_max := max(e1.var_max, e2.var_max)

{Post: e = e1 \/ e2}
```

1.6.4 Implementación de `niega_expresión`

Justificación: Como la negación es una operación unaria, plantamos el operador (-3) sobre un árbol (el de la expresión dato) colocando, por convenio, éste como hijo izquierdo, y un árbol nulo como hijo derecho.

Código:

```
función niega_expresión(e1: expresión) ret e: expresión;
{Pre: cierto}

   e.arb_exp := plantar (-3, e1.arb_exp, a_n);
   e.var_max := e1.var_max

{Post: e = !e1}
```

1.6.5 Implementación de consulta_max_var

Justificación: La consulta del índice máximo de una expresión es inmediata.

Código:

```
función consulta_max_var(e: expresión) ret n: natural;
{Pre: cierto}

   n:=e.var_max

{Post: n = índice máximo de las variables de e}
```

1.6.6 Implementación de satisface_expresión

Justificación: Para la operación `satisface_expresión` usamos una función privada que trabaje sólo con el árbol, ya que para realizar los cálculos no necesitamos saber cuál es la variable de mayor índice de los subárboles, que sería costosa de obtener.

La especificación de la operación auxiliar es

```
función satisface_aux(a: árbol de enteros; c: candidato) ret b:booleano;
{Pre: a es el árbol de una expresión válida y c contiene al menos tantos
valores como el índice máximo de las variables de a}

{Post: b = c satisface la expresión contenida en a}
```

Código de `satisface_expresión`:

```
función satisface_expresión(e: expresión; c: candidato) ret b: booleano;
{Pre: e es una expresión válida y c contiene al menos tantos valores como
el índice máximo de las variables de e}

   b:=satisface_aux(e.arb_exp,c)

{Post: b = c satisface e}
```

Dicha llamada es correcta, pues la precondición de la función original implica la de la auxiliar. El resultado obtenido es el deseado, ya que la postcondición de la auxiliar implica la de la original.

1.6.7 Implementación de satisface_aux

Justificación: Realizaremos un diseño recursivo. El único caso directo es cuando la raíz de a es positiva, lo que significa que la expresión consiste solamente en una variable. En tal caso, la expresión se satisface si el valor que le proporciona el candidato a dicha variable es cierto, para lo que disponemos de la correspondiente operación. El resto de casos son recursivos, ya que requieren saber si se satisfacen las expresiones contenidas en los hijos de a. Nos queda la siguiente alternativa.

```
x:=raíz(a);
si x>0 -> b:=consulta_valor(c,x)
[] x=-1 -> ?
[] x=-2 -> ?
[] x=-3 -> ?
fsi
```

Por supuesto, la alternativa es correcta porque los nodos del árbol sólo pueden tener esos valores, al tratarse de una expresión válida. La consulta a la raíz de a también es correcta, puesto que para no serlo a debería ser el árbol vacío, pero éste no representa a ninguna expresión válida.

Si $x = -1$, la expresión contenida en a es una conjunción. Por lo tanto, será satisfecha por c si lo son las dos expresiones contenidas en sus subárboles. De ahí que necesitemos realizar sendas llamadas recursivas y combinar adecuadamente sus resultados. Las llamadas son correctas ya que sus parámetros están bien definidos (a no es nulo) y sus precondiciones se cumplen porque los parámetros son subárboles del árbol inicial y por tanto conservan la precondición original. El decrecimiento de los parámetros se demuestra utilizando la operación *tamaño* de un árbol.

```
[] x=-1 -> b:=satisface_aux(hi(a),c);
           b1:=satisface_aux(hd(a),c);
           {HI: b= c satisface hi(a) /\ b1= c satisface hd(a}}
           b:= b /\ b1
```

Si $x = -2$, la expresión contenida en a es una disyunción. Por lo tanto, para ser satisfecha por c es suficiente con que lo sea una de las expresiones contenidas en sus subárboles. El código resultante y el razonamiento de su correción son similares a los de la rama anterior.

```
[] x=-2 -> b:=satisface_aux(hi(a),c);
           b1:=satisface_aux(hd(a),c);
           {HI: b= c satisface hi(a) /\ b1= c satisface hd(a}}
           b:= b \/ b1
```

Por último, si $x = -3$, la expresión contenida en a es una negación. Por el convenio de la contructora correspondiente, dicha expresión será satisfecha por c si la contenida su hijo izquierdo no lo es. El resto del razonamiento es el mismo que en los casos anteriores y da lugar al código

```
   [] x=-3 -> b:=satisface_aux(hi(a),c);
              {HI: b= c satisface hi(a)}
              b:= !b
```

Notad que el decrecimiento de los parámetros se demuestra empleando la operación *tamaño* de un árbol en las tres ramas.

Código:

```
función satisface_aux(a: árbol de enteros; c: candidato) ret b:booleano;
{Pre: a es el árbol de una expresión válida y c contiene al menos tantos
valores como el índice máximo de las variables de a}

   x:=raíz(a);
   si x>0 -> b:=consulta_valor(c,x)
   [] x=-1 -> b:=satisface_aux(hi(a),c);
              b1:=satisface_aux(hd(a),c);
              {HI: b= c satisface hi(a) /\ b1= c satisface hd(a}}
              b:= b /\ b1
   [] x=-2 -> b:=satisface_aux(hi(a),c);
              b1:=satisface_aux(hd(a),c);
              {HI: b= c satisface hi(a) /\ b1= c satisface hd(a}}
              b:= b \/ b1
   [] x=-3 -> b:=satisface_aux(hi(a),c);
              {HI: b= c satisface hi(a)}
              b:= !b
   fsi

{Post: b = c satisface la expresión contenida en a}
```

1.7 Implementación del módulo `Candidato`

En esta sección supondremos que nos han pedido justificar **formalmente** el diseño de las operaciones recursivas o iterativas del tipo `candidato`. Las demás serán diseñadas informalmente.

Para representar los candidatos tengamos en cuenta lo siguiente: si identificamos el valor *falso* con el dígito 0 y el valor *cierto* con el 1, entonces las posibles combinaciones de valores para n variables son los números binarios de n dígitos, es decir, todos los números entre 0 y $2^n - 1$. Por ejemplo, si tenemos 3 variables los posibles valores son:

0	0	0, que corresponde a 0,		
0	0	1	”	1,

0	1	0	"	2,
0	1	1	"	3,
1	0	0	"	4,
1	0	1	"	5,
1	1	0	"	6,
1	1	1	"	7

Notad que el valor decimal de cada combinación es igual al de la anterior más uno.

Así, podemos representar un candidato mediante un vector de ceros y unos. Si la posición i de un candidato es cero, significa que le asocia a la variable i-ésima el valor *falso* y lo mismo con uno y *cierto*. Además, aunque tenemos acotado el número de variables por MAX_VAR, el número real de variables de un candidato puede ser menor, así que necesitamos un natural que indique exactamente cuántas variables contiene.

```
Módulo Candidato;

{ Tipo de módulo: datos }

Implementación;

Const MAX_VAR =10;

Tipo candidato =
   tupla
     val : vector [1..MAX_VAR] de naturales
     tam : natural
   ftupla;

{El campo val contiene los valores de las variables. El campo tam
indica el número de éstos}
```

1.7.1 Implementación de `primer_candidato`

Justificación: Esta función retorna el candidato que representa al número cero con `num_var` dígitos. La asignación del campo `tamaño` es inmediata. Para el campo `vector` recurrimos a una función auxiliar `cero`, que especificamos a continuación y retomaremos más adelante para obtener su versión formal, que devuelve un vector con ceros en sus primeras `num_var` posiciones.

```
función cero (num_var: nat) ret v:vector;
{Pre: 1 =< num_var =< MAX_VAR }

{Post: v es un vector cuyas primeras num_var posiciones valen 0}
```

Código de `primer_candidato`:

```
función primer_candidato (num_var: nat) ret c:candidato;
{Pre: 1 =< num_var =< MAX_VAR }

   c.tam:=num_var;
   c.val:=cero(num_var)

{Post: c es el primer candidato para una expresión de num_var variables}
```

1.7.2 Implementación de `siguiente_candidato`

Justificación: Construimos el candidato correspondiente a sumar 1 al valor numérico del candidato de entrada. El campo `tamaño` es el mismo que el de éste y el vector se obtiene con una operación auxiliar `sumar_uno` que, de nuevo, especificamos ahora informalmente y completaremos más adelante.

```
función sumar_uno (w:vector; n:natural) ret v:vector;
{Pre: w tiene al menos un cero entre sus n primeras posiciones}

{Post: v es el vector que representa al resultado de sumar 1 al número
representado por w }
```

Código de `siguiente_candidato`:

```
función siguiente_candidato (c1:candidato) ret c2:candidato;
{Pre: c1 no es el último candidato para una expresión del número
correspondiente de variables}

   c2.tam:=c1.tam
   c2.val:=sumar_uno(c1.var,c1.tam)

{Post: c2 es el siguiente candidato a c1 }
```

1.7.3 Implementación de `último_candidato`

Justificación: El último candidato para un cierto número de variables es aquel que tiene todas sus posiciones relevantes ocupadas por unos. Usaremos una función auxiliar `buscar` para averiguar lo contrario, es decir, si hay un cero entre dichas posiciones.

```
función buscar (v:vector; n:natural; x: natural) ret b:booleano;
{Pre: 1 <= n <= MAX_VAR}

{Post: b indica si el valor x aparece entre las n primeras posiciones de v}
```

Código de `último_candidato`**:**

```
función último_candidato (c:candidato) ret b:booleano;
{Pre: cierto}

   b:= !buscar(c.val,c.tam,0)

{Post: b = c es el último candidato para una expresión del número
correspondiente de variables }
```

1.7.4 Implementación de `consultar_valor`

Justificación: Consultamos si la posición correspondiente del vector vale 1.

Código:

```
función consultar_valor (c:candidato; i:natural) ret b:booleano;
{Pre: 1 <= i <= número de valores de c}

   b:=(c.val[i] = 1)

{Post: b =  valor que c asocia a la i-ésima variable}
```

1.7.5 Resolución formal de las tres últimas funciones

Para derivar formalmente las tres funciones restantes necesitamos traducir su especificación a un lenguaje formal. Como sus parámetros son números y vectores, introduciremos cuantificadores para expresar formalmente sus asertos.

La función *cero*

La función *cero* devuelve un vector cuyos n primeros valores son cero.

$$\begin{array}{l} \textbf{funcion}\ \text{cero}(n:\text{natural})\ \textbf{retorna}\ v:\text{vector} \\ \quad \{1 \le n \le MAX\_VAR\} \\ \\ \quad \{\forall \alpha : 1 \le \alpha \le n : v[\alpha] = 0\} \end{array}$$

Aquí nada impide un diseño recursivo directo, ya que disponemos del parámetro n. Un caso directo es $n = 0$, pues entonces cualquier vector satisface la postcondición.

$$\begin{array}{ll} & \forall \alpha : 1 \le \alpha \le n : v[\alpha] = 0 \\ \Leftrightarrow & (n = 0) \\ & \forall \alpha : 1 \le \alpha \le 0 : v[\alpha] = 0 \\ \Leftrightarrow & (\text{dominio nulo de un } \forall) \\ & \text{cierto} \end{array}$$

Si $n > 0$, podemos intentar una llamada recursiva con $n-1$ (que es un parámetro válido, cumple la precondición y es menor que n) y seguramente será sencillo completar los cálculos necesarios. La situación es

funcion cero(n : natural) **retorna** v : vector
  $\{1 \leq n \leq MAX\_VAR\}$

  si $n = 0 \longrightarrow$ continuar
  [] $n > 0 \longrightarrow v :=$ cero$(n-1)$;
    {HI: $\forall \alpha : 1 \leq \alpha \leq n-1 : v[\alpha] = 0\}$
    ?
  fsi

  $\{\forall \alpha : 1 \leq \alpha \leq n : v[\alpha] = 0\}$

Ahora obtenemos las instrucciones que faltan a partir de la postcondición.

  $\forall \alpha : 1 \leq \alpha \leq n : v[\alpha] = 0$
$\Leftrightarrow$ ($n > 0$, separamos el n-simo sumando)
  $\forall \alpha : 1 \leq \alpha \leq n-1 : v[\alpha] = 0 \ \wedge \ v[n] = 0$
$\Leftrightarrow$ (si no modificamos las $n-1$ primeras posiciones del v de HI)
  $v[n] = 0$

que se logra con la correspondiente asignación.

funcion cero(n : natural) **retorna** v : vector
  $\{1 \leq n \leq MAX\_VAR\}$

  si $n = 0 \longrightarrow$ continuar
  [] $n > 0 \longrightarrow v :=$ cero$(n-1)$;
    {HI: $\forall \alpha : 1 \leq \alpha \leq n-1 : v[\alpha] = 0\}$
    $v[n] := 0$
  fsi

  $\{\forall \alpha : 1 \leq \alpha \leq n : v[\alpha] = 0\}$

La función *sumar_uno*

La función *sumar_uno* suma una unidad al número binario representado por los n primeros elementos de un vector, siempre que éstos no sean todos iguales a 1.

funcion sumar_uno(w : vector; n : natural) **retorna** v : vector
  $\{\exists \alpha : 1 \leq \alpha \leq n : w[\alpha] = 0 \ \wedge \ 1 \leq n \leq MAX\_VAR\}$

  $\{\Sigma \alpha : 1 \leq \alpha \leq n : w[\alpha]2^{\alpha-1} + 1 = \Sigma \alpha : 1 \leq \alpha \leq n : v[\alpha]2^{\alpha-1}\}$

Para derivarla emplearemos una inmersión. Si bien esto no es estrictamente necesario, podéis intentar un diseño recursivo directo y comprobaréis que la solución es ineficiente, con lo que de todas formas habría que realizar a continuación una inmersión de eficiencia.

Busquemos una versión débil de su postcondición:

$\sum_{\alpha=1}^{n} w[\alpha]2^{\alpha-1} + 1 = \sum_{\alpha=1}^{n} v[\alpha]2^{\alpha-1}$
$\Leftrightarrow$ (substitución de 1 por i)
$\sum_{\alpha=i}^{n} w[\alpha]2^{\alpha-i} + 1 = \sum_{\alpha=i}^{n} v[\alpha]2^{\alpha-i} \ \wedge \ i = 1$
$\Rightarrow$ (debilitando y añadiendo la correspondiente condición de dominio)
$\sum_{\alpha=i}^{n} w[\alpha]2^{\alpha-i} + 1 = \sum_{\alpha=i}^{n} v[\alpha]2^{\alpha-i} \ \wedge \ 1 \le i \le n \ \wedge \ \exists\alpha : i \le \alpha \le n : w[\alpha] = 0$

La función de inmersión, que llamaremos *sumar_uno_aux*, tendrá un dato adicional, la variable i. Además, su precondición debe reforzarse puesto que la suma se realiza sólo para los dígitos que van del i-ésimo al n-ésimo, se necesita que alguno de ellos sea cero.

funcion sumar_uno_aux(w : vector; n, i : natural) **retorna** v : vector
$\{1 \le n \le MAX\_VAR \ \wedge \ \exists\alpha : i \le \alpha \le n : w[\alpha] = 0 \ \wedge \ 1 \le i \le n\}$

$\{\sum_{\alpha=i}^{n} w[\alpha]2^{\alpha-i} + 1 = \sum_{\alpha=i}^{n} v[\alpha]2^{\alpha-i}\}$

En estas condiciones, es sencillo probar que si $i = 1$ entonces las dos precondiciones coinciden y también lo hacen las dos postcondiciones, lo que permite completar el diseño de *sumar_uno*:

funcion sumar_uno(w : vector; n : natural) **retorna** v : vector
$\{1 \le n \le MAX\_VAR \ \wedge \ \exists\alpha : 1 \le \alpha \le n : w[\alpha] = 0\}$

$v :=$ sumar_uno_aux$(w, n, 1)$

$\{\sum_{\alpha=1}^{n} w[\alpha]2^{\alpha-n} + 1 = \sum_{\alpha=1}^{n} v[\alpha]2^{\alpha-n}\}$

Pasemos a *sumar_uno_aux*. Como $i \le n$, podemos separar el i-ésimo elemento del sumatorio:

$\sum_{\alpha=i}^{n} w[\alpha]2^{\alpha-i} + 1 = \sum_{\alpha=i}^{n} v[\alpha]2^{\alpha-i}$
$\Leftrightarrow$ ($i \le n$; sacar el i-ésimo sumando; asociatividad)
$(w[i] + 1) + \sum_{\alpha=i+1}^{n} w[\alpha]2^{\alpha-i} = v[i] + \sum_{\alpha=i+1}^{n} v[\alpha]2^{\alpha-i}$

Hay un caso claro: si $w[i] = 0$, construir un vector v que satisfaga esa condición es sencillo, pues sirve el propio w, cambiando su posición i-ésima a 1. Podemos usar las asignaciones

$v := w;$
$v[i] := 1$

aunque en realidad es suficiente con asignar las posiciones $i+1$-ésima hasta la n-ésima.

Si $w[i] = 1$ podemos escribir

$(w[i] + 1) + \sum_{\alpha=i+1}^{n} w[\alpha]2^{\alpha-i} = v[i] + \sum_{\alpha=i+1}^{n} v[\alpha]2^{\alpha-i}$
$\Leftrightarrow$ $(w[i] = 1)$
$2 + \sum_{\alpha=i+1}^{n} w[\alpha]2^{\alpha-i} = v[i] + \sum_{\alpha=i+1}^{n} v[\alpha]2^{\alpha-i}$

$\Leftrightarrow$ (factor común 2; conmutatividad)
$2(\sum_{\alpha=i+1}^{n} w[\alpha]2^{\alpha-(i+1)} + 1) = v[i] + 2(\sum_{\alpha=i+1}^{n} v[\alpha]2^{\alpha-(i+1)})$

Ahora bien, esa igualdad se parece mucho a la postcondición de la llamada

$$v := \text{sumar\_uno\_aux}(w, n, i+1)$$

Tal llamada es válida, porque

- $i + 1$ es un número natural
- Su precondición viene implicada por la de la llamada anterior y el hecho de que $w[i] = 1$. En efecto, si ha de haber alguna posición que valga 0 entre i y n y no es i, deberá ser alguna de las siguientes.
- Como veremos al final, los parámetros decrecen en un cierto preorden.

La postcondición obtenida, aplicando la hipótesis de inducción, es

$$\sum_{\alpha=i+1}^{n} w[\alpha]2^{\alpha-(i+1)} + 1 = \sum_{\alpha=i+1}^{n} v[\alpha]2^{\alpha-(i+1)}$$

Podemos intentar resolver esta rama partiendo de dicha llamada. Queda la situación

$\{\text{Pre} \wedge w[i] = 1\}$

$v := \text{sumar\_uno\_aux}(w, n, i+1);$

$\{\text{HI: } \sum_{\alpha=i+1}^{n} w[\alpha]2^{\alpha-(i+1)} + 1 = \sum_{\alpha=i+1}^{n} v[\alpha]2^{\alpha-(i+1)}\}$

?

$\{\text{Post: } 2(\sum_{\alpha=i+1}^{n} w[\alpha]2^{\alpha-(i+1)} + 1) = v[i] + 2(\sum_{\alpha=i+1}^{n} v[\alpha]2^{\alpha-(i+1)})\}$

Ahora bien, si multiplicamos por 2 a ambos lados del aserto HI, observamos que éste coincide con el inferior, salvo por la presencia de $v[i]$. Para que coincidan del todo, $v[i]$ debe de valer 0, lo que se logra con una simple asignación. La función queda, pues,

funcion sumar_uno_aux(w : vector; n, i : natural) **retorna** v : vector
$\{1 \leq n \leq MAX\_VAR \ \wedge \ \exists\alpha : i \leq \alpha \leq n : w[\alpha] = 0 \ \wedge \ 1 \leq i \leq n\}$

si $w[i] = 0 \longrightarrow v := w;$
$\qquad v[i] := 1$
[] $w[i] = 1 \longrightarrow v := \text{sumar\_uno\_aux}(w, n, i+1);$
$\qquad v[i] := 0$
fsi

$\{\sum_{\alpha=i}^{n} w[\alpha]2^{\alpha-i} + 1 = \sum_{\alpha=i}^{n} v[\alpha]2^{\alpha-i}\}$

Para demostrar que los parámetros de las llamadas recursivas decrecen en algún preorden, consideramos la función $\mathrm{t} = n - i$. En las condiciones de la rama recursiva su valor es un número natural y es evidente que decrece en cada llamada.

La función *buscar*

La función *buscar* busca un elemento entre los n primeros dígitos de un vector.

funcion buscar(v : vector; n, x : natural;) **retorna** b : booleano
$\{1 \leq n \leq MAX\_VAR\}$

$\{b = \exists\alpha : 1 \leq \alpha \leq n : v[\alpha] = x\}$

Realizaremos un diseño iterativo. Es un problema clásico pero no está de más ver una solución formal (de las muchas posibles).

Obtenemos un candidato a invariante debilitando la postcondición. Notad que no es necesaria ninguna variable auxiliar, ya que podemos decrementar la n. Eso sí, entonces en la postcondición hay que hacer referencia a su valor inicial, que llamamos N:

$$b = \exists\alpha : 1 \leq \alpha \leq N : v[\alpha] = x$$
$\Rightarrow$ (debilitamos y añadimos la condición de dominio)
$$b = \exists\alpha : n < \alpha \leq N : v[\alpha] = x \ \wedge \ 0 \leq n \leq N$$

Veamos qué iteración se obtiene a partir de este invariante. Repasemos algunos de los elementos necesarios:

- *Condición de continuación*

  Tomamos $n > 0$, ya que si $n = 0$ se recupera la postcondición.

  $$b = \exists\alpha : n < \alpha \leq N : v[\alpha] = x \ \wedge \ 0 \leq n \leq N$$
  $\Leftrightarrow$ $(n = 0)$
  $$b = \exists\alpha : 0 < \alpha \leq N : v[\alpha] = x \ \wedge \ 0 \leq 0 \leq N$$
  $\Leftrightarrow$

  Post

- *Inicialización*

  Para inicializar el bucle sólo necesitamos poner b a *falso* ya que, de entrada, $n = N$ y el dominio del existencial queda vacío.

  $$b = \exists\alpha : n < \alpha \leq N : v[\alpha] = x \ \wedge \ 0 \leq n \leq N$$
  $\Leftrightarrow$ $(n = N)$
  $$b = \exists\alpha : N < \alpha \leq N : v[\alpha] = x \ \wedge \ 0 \leq 0 \leq N$$
  $\Leftrightarrow$ (dominio nulo del $\exists$)
  $$b = \text{falso}$$

- *Función de cota*

  Empleamos el propio n, ya que éste comienza valiendo N, que es mayor o igual que 1 por la precondición, y acaba valiendo 0, con lo que su valor habrá de decrecer.

Tras estas consideraciones, ya tenemos un esquema para la funcion, ya que una vez introducido el decrecimiento de n sólo queda por derivar una asignación sobre b:

funcion buscar(v : vector; n, x : natural;) **retorna** b : booleano
$\{1 \le n \le MAX\_VAR \ \wedge \ n = N\}$

$b :=$ falso;
mientras $n > 0$ **hacer**
$b := E$;
$n := n - 1$
fmientras

$\{b = \exists \alpha : 1 \le \alpha \le N : v[\alpha] = x\}$

La expresión E ha de cumplir

$$\text{Pre} \ \wedge \ \text{I} \ \wedge \ n > 0 \Longrightarrow (\text{I}^n_{n-1})^b_E$$

Si desarrollamos la parte derecha de la implicación queda

$(\text{I}^n_{n-1})^b_E$
$\Leftrightarrow$ (sustitución)
$E = \exists \alpha : n - 1 < \alpha \le N : v[\alpha] = x \ \wedge \ 0 \le n - 1 \le N$
$\Leftrightarrow$ ($0 < n \le N$; el segundo conjuntando se cumple; se puede separar el elemento n-ésimo del $\exists$)
$E = \exists \alpha : n < \alpha \le N : v[\alpha] = x \ \vee \ v[n] = x$
$\Leftrightarrow$ (I)
$E = b \ \vee \ v[n] = x$

Así, ya tenemos el valor de E. Notemos ademas que la posición n de v está definida por $n > 0$ (protección) y $n \le N \le MAX\_VAR$ (I y precondición).

Con esto queda completo el programa. Si además deseamos que éste termine en el momento de encontrar el valor x, habremos de añadir a la condición de continuación la protección $\neg b$.

funcion buscar(v : vector; n, x : natural;) **retorna** b : booleano
$\{1 \le n \le MAX\_VAR \ \wedge \ n = N\}$

$b :=$ falso;
mientras $n > 0 \wedge \neg b$ **hacer**
$b := v[n] = x$;
$n := n - 1$
fmientras

$\{b = \exists \alpha : 1 \le \alpha \le N : v[\alpha] = x\}$

La única demostración adicional requerida es que la negación de la condición de continuación implica la postcondición, ya que ahora el antecedente es más débil. De hecho, es suficiente demostrar que $\mathrm{I} \ \wedge \ b \Longrightarrow \mathrm{Post}$. En cualquier caso, esa propiedad se cumple ya que si b es cierta, el existe desde n hasta N es cierto y eso implica que también lo es desde 1 hasta N.

Notad también que la asignación sobre b se puede simplificar ya que ahora si se entra en el bucle su valor es *falso*.

1.7.6 Comentario sobre eficiencia

La operación *último* se puede implementar de manera más eficiente (es decir, sin recorrer el vector cada vez), si para cada candidato guardamos también su valor numérico asociado. De ese modo, un candiato de n variables es el último si su valor asociado es $2^n - 1$. Ello afectaría también a las operaciones constructoras `primero` y `siguiente`.

Notad que en una práctica de verdad, habría que desarrollar completamente la nueva solución para que cuente como tal.