Implementación de los números enteros positivos.

Para empezar hemos de comprender la estructura *multiplicativa* de los positivos. A diferencia del caso de los naturales donde su estructura aditiva nos permite construir todos los números con una sola operacion básica (sucesor) obteniendo así un representación unaria, hay ahora dos operaciones básicas para construir todos los positivos a partir del uno: una consiste en multiplicar por 2 y sumar 1; la otra es multiplicar por 2. De esta forma dado un representante para el 1, todo positivo puede representarse de modo único con dos operaciones (representación binaria) aplicadas a ese inicial 1. Por ejemplo:

$$27 = 2 \times (2 \times (2 \times (2 \times (1) + 1)) + 1) + 1$$

Introduciremos pues los positivos como el siguiente tipo inductivo:

```
Inductive positive : Set :=
  xI : positive → positive | xO : positive → positive | xH : positive.
```

En la librería podemos ver como se define la suma y el producto de estos números. Nosotros nos centraremos en la representación del tipo de dato.

Podemos construir funciones sobre este nuevo tipo. Por ejemplo la función que suma uno a un positivo. La podemos programar directamente por *pattern matching*:

```
Fixpoint add_un (x:positive):positive :=
    match x with
              xI x' ⇒ xO (add_un x')
             |xO x' ⇒ xI x'
             |xH ⇒ xO xH
    end.
```

aunque podemos hacerlo utilizando el esquema recursivo asociado a *positive*.

Ejercicio 1: Programar *add_un* usando *positive_rec*.

```
Eval compute in add_un (xO xH).
```

Este es otro ejemplo:

```
Fixpoint Pdouble_minus_one (x:positive) : positive :=
  match x with
  | xI x' ⇒ xI (xO x')
```

```
| xO x' ⇒ xI (Pdouble_minus_one x')
| xH ⇒ xH
end.
```

Podemos comprobar propiedades de estas funciones:

```
Lemma add_one_pd:
  ∀ p:positive, add_un (Pdouble_minus_one p) = xO p.
 Proof.
 intro x; induction x as [x IHx| x| ]; simpl in ⊢ ×; try rewrite IHx;
  auto.
 Qed.

Lemma Pd_add_un :
  ∀ p:positive, Pdouble_minus_one (add_un p) = xI p.
 Proof.
 intro x; induction x as [x IHx| x| ]; simpl in ⊢ ×; try rewrite IHx;
  reflexivity.
 Qed.

Lemma add_un1: ∀ x y, (add_un x)=(add_un y)-¿ x=y.
intros.
cut ((Pdouble_minus_one (add_un x))=(Pdouble_minus_one (add_un y))).
Show 2.
2:rewrite H;auto.
intros.
rewrite (Pd_add_un x) in H0.
rewrite (Pd_add_un y) in H0.
injection H0.
auto.
Qed.
```

A continuación definimos una función que nos convierte un positivo en "su valor" como natural.

```
Fixpoint pos2nat (p:positive):nat:=
match p with
 xI x' ⇒S (2*(pos2nat x'))
|xO x' ⇒ 2*(pos2nat x')
|xH ⇒ 1
end.
```

Ejercicio 2: Construir pos2nat usando el bucle de prueba. Una vez construida, imprimirla y compararla con esta y comprobar que funcionan igual.

Los lemas siguientes, aunque triviales, resultan útiles.

```
Lemma pos2nat1:∀ x':positive,pos2nat
 (xI x')=S (2*(pos2nat x')).
simpl;auto.
Qed.
Lemma pos2nat2:∀ x':positive,pos2nat (xO x')= 2*(pos2nat x').
simpl;auto.
Qed.
Lemma pos2nat3:pos2nat xH=1.
simpl;auto.
Qed.

Hint Rewrite pos2nat1 pos2nat2 pos2nat3:core.

Eval compute in pos2nat (xI (xI (xO (xI xH)))).
Require Export Lt.

Lemma pos2natlt: ∀ p:positive, 0¡pos2nat p .
intros.
induction p;autorewrite with core.
auto with v62.
induction IHp.
auto with v62.
replace (2× S m) with (S (S (2×m))).
auto with v62.
simpl.
auto with v62.
auto with v62.
Qed.

Require Export Arith.
Lemma aux0:∀ x,x+0=x.
auto.
Defined.
```

Este es un lema auxiliar que necesitamos porque estamos operando sin el *Scope* del tipo de la librería. Como ya se ha dicho, en uso ordinario todos estos resultados son de prueba automática

```
Lemma auxSS:∀ x, (S x)+(S x)=S (S (x+x)).
intro x.
replace (S x+(S x)) with (S (x + (S x))).
apply eq_S.
auto with arith.
auto with arith.
Defined.

Lemma pos2natS:∀ x:positive, pos2nat (add_un x) = S (pos2nat x).
induction x.
simpl.
rewrite IHx.
rewrite aux0.
rewrite (auxSS (pos2nat x)).
auto.
simpl.

rewrite aux0.
auto.
auto.
Qed.
```

He aquí otro resultado que hemos de probar *a mano*. Es importante el orden ($0 <> n$ y no $n <> 0$, por ejemplo) para que el tipo de las funciones de la librería *Arith* (por ejemplo O_S) coincidan y todo sea más sencillo de probar:

```
Lemma mm:∀ n:nat, {0=n}+{0≠n}.
induction n.
left;auto.
right;apply O_S.
Defined.
```

Esta es, probablemente, la función más compleja de esta lección. Se trata de "pasar" de un natural (no nulo) a su representación como positivo. Nótese que el tipo de este programa es dependiente ($\forall n, 0 \neq n \rightarrow positive$):

```
Fixpoint nat2pos (n:nat) : 0≠n→positive:=
match n return 0≠n→positive with
|O ⇒ fun h:0≠0 ⇒ False_rec positive (h (refl_equal 0))
|(S p) ⇒ fun _ ⇒ match (mm p) with
                 |left _ ⇒ xH
                 |right H ⇒ add_un (nat2pos p H)
```

end

end.

Alternativamente podemos construirla usando el bucle de prueba y probar que ambas definiciones son equivalentes:

Definition *nat2pos'*:∀ *n*,0≠*n*→*positive.*
induction n.
intros abs.
absurd (0≠0);*auto.*
case (*mm n*).
intros.
exact xH.
intros.
exact (*add_un* (*IHn n0*)).
Defined.

Print *nat2pos'.*

Lemma *nat2pos1*:∀ *n*,*nat2pos* (*S* (*S n*)) (*O_S* (*S n*)) = *add_un* (*nat2pos* (*S n*) (*O_S n*)).
intros;*simpl.*
auto.
Defined.

Lemma *nat2pos'1*:∀ *n*,*nat2pos'* (*S* (*S n*)) (*O_S* (*S n*)) = *add_un* (*nat2pos'* (*S n*) (*O_S n*)).
intros;*simpl.*
auto.
Defined.

Lemma *compr*:∀ *n*,(*nat2pos* (*S n*) (*O_S n*))=(*nat2pos'* (*S n*) (*O_S n*)).
induction n.
simpl;*auto.*
rewrite (*nat2pos1 n*).
rewrite (*nat2pos'1 n*).
rewrite IHn.
auto.
Qed.

Eval compute in (*nat2pos* 6 (*O_S* 5)).
Eval compute in (*nat2pos* 27 (*O_S* 26)).
Eval compute in (*nat2pos* 0).

Algunas Propiedades

Eval compute in pos2nat (nat2pos 6 (O_S 5)).
Eval compute in (let n := (pos2nat (xI (xI (xO (xI xH))))) in
nat2pos n (O_S (pred n))).

Theorem *equiv1*:∀ *n*, *pos2nat (nat2pos (S n) (O_S n))=S n.*
induction n.
simpl;*auto.*
rewrite (nat2pos1 n).
rewrite (pos2natS (nat2pos (S n) (O_S n))).
apply eq_S.
auto.
Qed.

Lemma *posS*:∀ *p*, {*n*:*nat*| (*pos2nat p*= *S n*)}.
induction p.
case IHp;*intros*;*clear IHp.*
simpl.
rewrite e.
∃ (*S x* +(*S x*)).
auto with arith.
case IHp;*intros*;*clear IHp.*
simpl.
rewrite e.
∃ (*S* (*x*+*x*)).
simpl.
apply eq_S.
Check plus_Sn_m.
rewrite ← (*plus_Sn_m x x*).
Check plus_n_O.
rewrite ← (*plus_n_O x*).
auto with arith.
∃ 0.
simpl;*auto.*
Defined.

Lemma *posS'*: ∀ *p*, {*n*:*nat*| (*pos2nat p*= *S n*)}.
intro p.

Check equiv1.
$\exists$ (*pred* (*pos2nat p*)).
Check (*S_pred* (*pos2nat p*) 0 (*pos2natlt p*)) .
rewrite $\leftarrow$ (*S_pred* (*pos2nat p*) 0 (*pos2natlt p*)).
auto.
Qed.