Introducción a ML II

José de Jesús Lavalle Martínez
http://aleteya.cs.buap.mx/~jlavalle/

Benemérita Universidad Autónoma de Puebla
Facultad de Ciencias de la Computación
Licenciatura en Ciencias de la Computación
Fundamentos de Lenguajes de Programación
CCOS 255

Contenido

Invierte una lista primer intento

Probablemente su primer intento para invertir dos listas fue:

```
fun  ilw(nul) = nul
|    ilw(ht(h,t)) = ht(ilw(t),h);
```

Invierte una lista primer intento

Pero el intérprete de ML nos respondió de la siguiente manera:

```
- fun ilw(nul) = nul
|   ilw(ht(h,t)) = ht(ilw(t),h);
! Toplevel input:
! |   ilw(ht(h,t)) = ht(ilw(t),h);
!                       ^^^^^^
! Type clash: expression of type
!   'a Milist
! cannot have type
!   'a
! because of circularity
-
```

Invierte una lista

La forma correcta para invertir una lista es:

```
fun  inv(nul) = nul
|    inv(ht(h,t)) = uneList(inv(t), ht(h, nul));
```

Invierte una lista

Así obtenemos:

```
- fun inv(nul) = nul
|     inv(ht(h,t)) = uneList(inv(t), ht(h, nul));
> val 'a inv = fn : 'a Milist -> 'a Milist
- inv(ht(1, ht(2, ht(3, ht(4, nul)))));
> val it = ht(4, ht(3, ht(2, ht(1, nul)))) : int Milist
-
```

Listas en ML

```
- [1, 2, 3, 4];
> val it = [1, 2, 3, 4] : int list
- nil;
> val 'a it = [] : 'a list
- 1::nil;
> val it = [1] : int list
- 2::it;
> val it = [2, 1] : int list
- 3::it;
> val it = [3, 2, 1] : int list
- 1::2::3::nil;
> val it = [1, 2, 3] : int list
-
```

Invierte con parámetro acumulante I

```
[1, 2, 3, 4] -> [4, 3, 2, 1]

1::[2, 3, 4], nil -> [2, 3, 4], 1::nil
2::[3, 4], [1] -> [3, 4], 2::[1]
3::[4], [2, 1] -> [4], 3::[2, 1]
4::[], [3, 2, 1] -> [], 4::[3, 2, 1]
[], [4, 3, 2, 1] -> [4, 3, 2, 1]
```

Invierte con parámetro acumulante II

```
fun invpa(nil,pa) = pa
|   invpa(h::t, pa) = invpa(t, h::pa);
```

Invierte con parámetro acumulante II

```
- fun invpa(nil,pa) = pa
|   invpa(h::t, pa) = invpa(t, h::pa);
> val 'a invpa = fn : 'a list * 'a list -> 'a list
- invpa([1, 2, 3, 4], nil);
> val it = [4, 3, 2, 1] : int list
-
```

Invierte con parámetro acumulante III

```
fun inversa(l) =
  let
    fun invpa(nil, pa) = pa
    |   invpa(h::t, pa) = invpa(h, h::pa)
  in
    invpa(l, nil)
  end;
```

Invierte con parámetro acumulante III

```
- fun inversa(l) =
  let
    fun invpa(nil, pa) = pa
    |   invpa(h::t, pa) = invpa(t, h::pa)
  in
    invpa(l, nil)
  end;
> val 'a inversa = fn : 'a list -> 'a list
- inversa([1, 2, 3, 4]);
> val it = [4, 3, 2, 1] : int list
-
```

Invierte con parámetro acumulante IV

```
- inversa;
> val 'a it = fn : 'a list -> 'a list
- invpa;
! Toplevel input:
! invpa;
! ^^^^^
! Unbound value identifier: invpa
-
```

De Milist a list

```
fun  Mil2l(nul) = nil
|    Mil2l(ht(h,t)) = h::Mil2l(t);
```

De Milist a list

```
- fun Mil2l(nul) = nil
|   Mil2l(ht(h,t)) = h::Mil2l(t);
> val 'a Mil2l = fn : 'a Milist -> 'a list
- Mil2l(ht(1, ht(2, ht(3, ht(4, nul)))));
> val it = [1, 2, 3, 4] : int list
-
```

ales I

Queremos modelar en ML a los naturales.

ales I

Queremos modelar en ML a los naturales.

1. 0 es un natural,
2. si n es un natural entonces $s(n)$ es un natural,
3. n es un natural sólo si se obtiene por las reglas 1 y 2 anteriores.

ales I

Queremos modelar en ML a los naturales.

1. 0 es un natural,
2. si n es un natural entonces $s(n)$ es un natural,
3. n es un natural sólo si se obtiene por las reglas 1 y 2 anteriores.

```
datatype ales = z | s of ales;
```

ales II

```
- datatype ales = z | s of ales;
> New type names: =ales
  datatype ales = (ales,{con s : ales -> ales, con z : ales})
  con s = fn : ales -> ales
  con z = z : ales
- z;
> val it = z : ales
- s z;
> val it = s z : ales
- s(s z);
> val it = s(s z) : ales
- s(s(s z));
> val it = s(s(s z)) : ales
-
```

Alto orden, funciones como parámetros I

```
fun aplica(nil, f) = nil
|   aplica(h::t, f:int->int) = f(h)::aplica(t, f);
```

```
- fun aplica(nil, f) = nil
|   aplica(h::t, f:int->int) = f(h)::aplica(t, f);
> val aplica = fn : int list * (int -> int) -> int list
-
```

Alto orden, funciones como parámetros I

```
fun sqr n = n*n;
```

```
- fun sqr n = n*n;
> val sqr = fn : int -> int
- aplica([1, 2, 3, 4], sqr);
> val it = [1, 4, 9, 16] : int list
-
```

Alto orden, funciones como parámetros II

Lo podemos hacer mejor:

```
fun aplicag(nil, f) = nil
|   aplicag(h::t, f) = f(h)::aplicag(t, f);
```

Alto orden, funciones como parámetros II

Lo podemos hacer mejor:

```
fun aplicag(nil, f) = nil
|   aplicag(h::t, f) = f(h)::aplicag(t, f);
```

```
fun aplicag(nil, f) = nil
|   aplicag(h::t, f) = f(h)::aplicag(t, f);
> val ('a, 'b) aplicag = fn : 'a list * ('a -> 'b) -> 'b list
```

Alto orden, funciones como parámetros II

```
- aplicag([1, 2, 3, 4], sqr);
> val it = [1, 4, 9, 16] : int list
- ceil;
> val it = fn : real -> int
- aplicag([1.1, 2.2, 3.3, 4.4], ceil);
> val it = [2, 3, 4, 5] : int list
- aplicag([1.1, 2.2, 3.3, 4.4], floor);
> val it = [1, 2, 3, 4] : int list
- fun sqrr n:real = n*n;
> val sqrr = fn : real -> real
- aplicag([1.1, 2.2, 3.3, 4.4], sqrr);
> val it = [1.21, 4.84, 10.89, 19.36] : real list
-
```

Ejercicios

En los ejercicios sólo puede definir las funciones hasta ahora definidas y las que vaya definiendo, por ejemplo, para `multiplicales` puede usar `sumales`, para `exponenciales` puede usar `multiplicales`.

1. Defina en ML las siguientes funciones:
   1. `ultimo = fn:'a list -> 'a`, que obtiene el último elemento de una lista.
   2. `long = fn:'a list -> int`, que obtiene la longitud de una lista.
   3. `nesimo = fn:'a list * int -> 'a`, que obtiene el n-ésimo elemento de una lista.
2. Utilizando la definición del tipo ales defina las siguientes funciones:
   1. `sumales = fn:ales * ales -> ales`, que suma dos elementos de `ales`.
   2. `multiplicales = fn:ales * ales -> ales`, que multiplica dos elementos de `ales`.
   3. `exponenciales = fn:ales * ales -> ales`, que exponencia dos `alesl`, es decir, `exponenciales(m, n)` $= m^n$.

Ejemplo de exponenciales I

```
exponenciales(2, 3) = 8
exponenciales(2, 3) = multiplicales(2, exponenciales(2, 2)) =
multiplicales(2, multiplicales(2, exponenciales(2, 1))) =
multiplicales(2, multiplicales(2,
multiplicales(2, exponenciales(2, 0)))) =
multiplicales(2, multiplicales(2, multiplicales(2, 1))) =
multiplicales(2, multiplicales(2, 2)) =
multiplicales(2, 4) = 8
```

Ejemplo de exponenciales II

```
exponenciales(s(s z), s(s(s z)))
```

Ejemplo de sumales

Con sumales prueben 23456 + 34567

```
ales2int = fn : ales -> int
int2ales = fn : int -> ales
sumales(int2ales(23456), int2ales(34567))

- int2ales(23456);
> val it = s(s(s(s(s(s(s(s(s(s(s(s(s(s(s(s(s(s(s(s#
                                    ))))))))))))))))))) : ales
- ales2int(it);
> val it = 23456 : int
-
```

sumales

```
fun sumales(z, n) = n
|   sumales(s m, n) =  s(sumales(m, n));
```

```
sumales(s(s z), s(s(s z))) = s(sumales(s z, s(s(s z)))) =
s(s(sumales(z, s(s(s z))))) = s(s(s(s(s z))))
```