Introducción a ML I

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

El mecánsmo para hacer cálculos en ML I

Para hacer un cálculo en ML requerimos:

1. Evaluar una función (puede ser una función interconstruida en el lenguaje o definida por el usuario).

El mecánsmo para hacer cálculos en ML I

Para hacer un cálculo en ML requerimos:

1. Evaluar una función (puede ser una función interconstruida en el lenguaje o definida por el usuario).
2. Las funciones tienen argumentos, los cuales deben ser de algún tipo (puede ser un tipo interconstruido en el lenguaje o definido por el usario).

El mecánsmo para hacer cálculos en ML I

Para hacer un cálculo en ML requerimos:

1. Evaluar una función (puede ser una función interconstruida en el lenguaje o definida por el usuario).
2. Las funciones tienen argumentos, los cuales deben ser de algún tipo (puede ser un tipo interconstruido en el lenguaje o definido por el usario).
3. Como ejemplos, tenemos todas las funciones que definimos en la primera sesión de introducción a ML, en ella construimos funciones sobre los números enteros.

El mecánsmo para hacer cálculos en ML I

Para hacer un cálculo en ML requerimos:

1. Evaluar una función (puede ser una función interconstruida en el lenguaje o definida por el usuario).
2. Las funciones tienen argumentos, los cuales deben ser de algún tipo (puede ser un tipo interconstruido en el lenguaje o definido por el usario).
3. Como ejemplos, tenemos todas las funciones que definimos en la primera sesión de introducción a ML, en ella construimos funciones sobre los números enteros.
4. Los enteros (`int`), reales (`real`), booleanos (`bool`), cadenas (`string`) y listas (`list`), entre otros, son tipos interconstruidos en ML.

El mecánsmo para hacer cálculos en ML II

1. Los tipos nos sirven para clasificar la información de acuerdo a sus características intrínsecas.

El mecánsmo para hacer cálculos en ML II

1. Los tipos nos sirven para clasificar la información de acuerdo a sus características intrínsecas.
2. Los tipos definidos por el usuario en ML son de tres variedades:

El mecánsmo para hacer cálculos en ML II

1. Los tipos nos sirven para clasificar la información de acuerdo a sus características intrínsecas.
2. Los tipos definidos por el usuario en ML son de tres variedades:
    1. finitos,

El mecánsmo para hacer cálculos en ML II

1. Los tipos nos sirven para clasificar la información de acuerdo a sus características intrínsecas.
2. Los tipos definidos por el usuario en ML son de tres variedades:
   1. finitos,
   2. potencialmente infinitos y

El mecánsmo para hacer cálculos en ML II

1. Los tipos nos sirven para clasificar la información de acuerdo a sus características intrínsecas.
2. Los tipos definidos por el usuario en ML son de tres variedades:
    1. finitos,
    2. potencialmente infinitos y
    3. genéricos.

El mecánsmo para hacer cálculos en ML II

1. Los tipos nos sirven para clasificar la información de acuerdo a sus características intrínsecas.
2. Los tipos definidos por el usuario en ML son de tres variedades:
   1. finitos,
   2. potencialmente infinitos y
   3. genéricos.
3. Un tipo en ML se define dando los **constructores** de ese tipo.

Tipos finitos I

Los tipos finitos se construyen dando explícitamente los elementos del tipo, los elementos son los constructores del tipo.

Tipos finitos I

Los tipos finitos se construyen dando explícitamente los elementos del tipo, los elementos son los constructores del tipo.

- Si queremos definir el tipo que representa las figuras de una carta americana hacemos lo siguiente:

```
datatype figuras = pica | trebol| diamante | corazon;
```

Tipos finitos I

Los tipos finitos se construyen dando explícitamente los elementos del tipo, los elementos son los constructores del tipo.

- Si queremos definir el tipo que representa las figuras de una carta americana hacemos lo siguiente:

```
datatype figuras = pica | trebol| diamante | corazon;
```

```
- datatype figuras = pica | trebol| diamante | corazon;
> New type names: =figuras
  datatype figuras =
  (figuras,
   {con corazon : figuras,
    con diamante : figuras,
    con pica : figuras,
    con trebol : figuras})
  con corazon = corazon : figuras
  con diamante = diamante : figuras
  con pica = pica : figuras
  con trebol = trebol : figuras
-
```

Tipos finitos I

Los tipos finitos se construyen dando explícitamente los elementos del tipo, los elementos son los constructores del tipo.

- Si queremos definir el tipo que representa las figuras de una carta americana hacemos lo siguiente:

```
datatype figuras = pica | trebol| diamante | corazon;
```

```
- datatype figuras = pica | trebol| diamante | corazon;
> New type names: =figuras
  datatype figuras =
  (figuras,
   {con corazon : figuras,
    con diamante : figuras,
    con pica : figuras,
    con trebol : figuras})
  con corazon = corazon : figuras
  con diamante = diamante : figuras
  con pica = pica : figuras
  con trebol = trebol : figuras
-

- corazon;
> val it = corazon : figuras
- diamante;
> val it = diamante : figuras
-
```

Tipos finitos II

- Si queremos definir los digitos de nuestros términos aritméticos:

```
datatype digi = d0 | d1 | d2 | d3 | d4 | d5 | d6 | d7 | d8 | d9;
```

Tipos finitos II

- Si queremos definir los digitos de nuestros términos aritméticos:

```
datatype digi = d0 | d1 | d2 | d3 | d4 | d5 | d6 | d7 | d8 | d9;

- datatype
digi = d0 | d1 | d2 | d3 | d4 | d5 | d6 | d7 | d8 | d9;
> New type names: =digi
  datatype digi =
  (digi,
   {con d0 : digi,
    con d1 : digi,
    con d2 : digi,
    con d3 : digi,
    con d4 : digi,
    con d5 : digi,
    con d6 : digi,
    con d7 : digi,
    con d8 : digi,
    con d9 : digi})
  con d0 = d0 : digi
  con d1 = d1 : digi
  con d2 = d2 : digi
  con d3 = d3 : digi
  con d4 = d4 : digi
  con d5 = d5 : digi
  con d6 = d6 : digi
  con d7 = d7 : digi
  con d8 = d8 : digi
  con d9 = d9 : digi
-
```

Tipos finitos II

- Si queremos definir los digitos de nuestros términos aritméticos:

```
datatype digi = d0 | d1 | d2 | d3 | d4 | d5 | d6 | d7 | d8 | d9;

- datatype
digi = d0 | d1 | d2 | d3 | d4 | d5 | d6 | d7 | d8 | d9;
> New type names: =digi
  datatype digi =
  (digi,
   {con d0 : digi,
    con d1 : digi,
    con d2 : digi,
    con d3 : digi,
    con d4 : digi,
    con d5 : digi,
    con d6 : digi,
    con d7 : digi,
    con d8 : digi,
    con d9 : digi})
  con d0 = d0 : digi
  con d1 = d1 : digi
  con d2 = d2 : digi
  con d3 = d3 : digi
  con d4 = d4 : digi
  con d5 = d5 : digi
  con d6 = d6 : digi
  con d7 = d7 : digi
  con d8 = d8 : digi
  con d9 = d9 : digi
-

- d5;
> val it = d5 : digi
-
```

Tipos potencialmente infinitos I

- La única forma que tenemos para definir de manera finita un tipo potencialmente infinito es mediante una definición recursiva.

Tipos potencialmente infinitos I

- La única forma que tenemos para definir de manera finita un tipo potencialmente infinito es mediante una definición recursiva.
- Debemos tener dos variedades de constructores:

Tipos potencialmente infinitos I

- La única forma que tenemos para definir de manera finita un tipo potencialmente infinito es mediante una definición recursiva.
- Debemos tener dos variedades de constructores:
  1. Los constructores básicos, aquellos que **no** hacen referencia recursiva al nombre del tipo que estamos definiendo.

Tipos potencialmente infinitos I

- La única forma que tenemos para definir de manera finita un tipo potencialmente infinito es mediante una definición recursiva.
- Debemos tener dos variedades de constructores:
  1. Los constructores básicos, aquellos que **no** hacen referencia recursiva al nombre del tipo que estamos definiendo.
  2. Los constructores recursivos, aquellos que **sí** hacen referencia recursiva al nombre del tipo que estamos definiendo.

Tipos potencialmente infinitos I

- La única forma que tenemos para definir de manera finita un tipo potencialmente infinito es mediante una definición recursiva.
- Debemos tener dos variedades de constructores:
  1. Los constructores básicos, aquellos que **no** hacen referencia recursiva al nombre del tipo que estamos definiendo.
  2. Los constructores recursivos, aquellos que **sí** hacen referencia recursiva al nombre del tipo que estamos definiendo.
- Por ejemplo, si queremos definir una lista de enteros, primero debemos tener claro como definir una lista, sabemos que:

Tipos potencialmente infinitos I

- La única forma que tenemos para definir de manera finita un tipo potencialmente infinito es mediante una definición recursiva.
- Debemos tener dos variedades de constructores:
  1. Los constructores básicos, aquellos que **no** hacen referencia recursiva al nombre del tipo que estamos definiendo.
  2. Los constructores recursivos, aquellos que **sí** hacen referencia recursiva al nombre del tipo que estamos definiendo.
- Por ejemplo, si queremos definir una lista de enteros, primero debemos tener claro como definir una lista, sabemos que:
  1. una lista está vacía, o

Tipos potencialmente infinitos I

- La única forma que tenemos para definir de manera finita un tipo potencialmente infinito es mediante una definición recursiva.
- Debemos tener dos variedades de constructores:
  1. Los constructores básicos, aquellos que **no** hacen referencia recursiva al nombre del tipo que estamos definiendo.
  2. Los constructores recursivos, aquellos que **sí** hacen referencia recursiva al nombre del tipo que estamos definiendo.
- Por ejemplo, si queremos definir una lista de enteros, primero debemos tener claro como definir una lista, sabemos que:
  1. una lista está vacía, o
  2. tiene un primer elemento que es un entero y un resto que a su vez es una lista de enteros.

Tipos potencialmente infinitos II

```
datatype intList = nuli | hti of int * intList;
```

Tipos potencialmente infinitos II

```
datatype intList = nuli | hti of int * intList;
```

```
- datatype intList = nuli | hti of int * intList;
> New type names: =intList
  datatype intList =
  (intList,
   {con hti : int * intList -> intList,
    con nuli : intList})
  con hti = fn : int * intList -> intList
  con nuli = nuli : intList
-
```

Tipos potencialmente infinitos II

```
datatype intList = nuli | hti of int * intList;
```

```
- datatype intList = nuli | hti of int * intList;
> New type names: =intList
  datatype intList =
  (intList,
   {con hti : int * intList -> intList,
    con nuli : intList})
  con hti = fn : int * intList -> intList
  con nuli = nuli : intList
-

- hti(1,hti(2, hti(3,nuli)));
> val it = hti(1, hti(2, hti(3, nuli))) : intList
-
```

Tipos genéricos I

Suponga que ahora queremos definir una lista de reales y una lista de cadenas, sus definiciones serían:

Tipos genéricos I

Suponga que ahora queremos definir una lista de reales y una lista de cadenas, sus definiciones serían:

```
datatype intList = nuli | hti of int * intList;
datatype realList = nulr | htr of real * realList;
datatype strList = nuls | hts of string * strList;
```

Tipos genéricos I

Suponga que ahora queremos definir una lista de reales y una lista de cadenas, sus definiciones serían:

```
datatype intList = nuli | hti of int * intList;
datatype realList = nulr | htr of real * realList;
datatype strList = nuls | hts of string * strList;
```

```
- htr(1.0,htr(2.0,htr(3.0,nulr)));
> val it = htr(1.0, htr(2.0, htr(3.0, nulr))) : realList
- hts("Hola",hts("cómo",hts("están",nuls)));
> val it = hts("Hola", hts("cómo", hts("están", nuls))) : strList
-
```

Tipos genéricos II

Es decir lo importante es la estructura, no importa el tipo de datos que van a contener las listas. Por ello ML nos permite definir tipos genéricos.

Tipos genéricos II

Es decir lo importante es la estructura, no importa el tipo de datos que van a contener las listas. Por ello ML nos permite definir tipos genéricos.

```
datatype 'a Milist = nul | ht of 'a * 'a Milist;
```

Tipos genéricos II

Es decir lo importante es la estructura, no importa el tipo de datos que van a contener las listas. Por ello ML nos permite definir tipos genéricos.

```
datatype 'a Milist = nul | ht of 'a * 'a Milist;
```

```
- datatype 'a Milist = nul | ht of 'a * 'a Milist;
> New type names: =Milist
  datatype 'a Milist =
  ('a Milist,
   {con 'a ht : 'a * 'a Milist -> 'a Milist,
    con 'a nul : 'a Milist})
  con 'a ht = fn : 'a * 'a Milist -> 'a Milist
  con 'a nul = nul : 'a Milist
-
```

Tipos genéricos III

Algunos ejemplos:

```
- ht(1, ht(2, ht(3, nul)));
> val it = ht(1, ht(2, ht(3, nul))) : int Milist
- ht(1.0, ht(2.0, ht(3.0, nul)));
> val it = ht(1.0, ht(2.0, ht(3.0, nul))) : real Milist
- ht("Hola", ht("cómo", ht("están", nul)));
> val it = ht("Hola", ht("cómo", ht("están", nul))) : string Milist
-
```

Tipos genéricos III

Algunos ejemplos:

```
- ht(1, ht(2, ht(3, nul)));
> val it = ht(1, ht(2, ht(3, nul))) : int Milist
- ht(1.0, ht(2.0, ht(3.0, nul)));
> val it = ht(1.0, ht(2.0, ht(3.0, nul))) : real Milist
- ht("Hola", ht("cómo", ht("están", nul)));
> val it = ht("Hola", ht("cómo", ht("están", nul))) : string Milist
-
```

Note que una vez que damos el primer elemento de la lista, el tipo genérico 'a se instancia con el tipo del primer elemento. Por ello tenemos el siguiente error.

Tipos genéricos III

Algunos ejemplos:

```
- ht(1, ht(2, ht(3, nul)));
> val it = ht(1, ht(2, ht(3, nul))) : int Milist
- ht(1.0, ht(2.0, ht(3.0, nul)));
> val it = ht(1.0, ht(2.0, ht(3.0, nul))) : real Milist
- ht("Hola", ht("cómo", ht("están", nul)));
> val it = ht("Hola", ht("cómo", ht("están", nul))) : string Milist
-
```

Note que una vez que damos el primer elemento de la lista, el tipo genérico 'a se instancia con el tipo del primer elemento. Por ello tenemos el siguiente error.

```
- ht(1, ht(1.0, nul));
! Toplevel input:
! ht(1, ht(1.0, nul));
!           ^^^
! Type clash: expression of type
!   real
! cannot have type
!   int
-
```

Unir dos listas I

1. Queremos definir una función en ML que una dos listas, es decir recibe las listas `ht(1, ht(2, nul))` y `ht(3, ht(4, nul))` y nos debe regresar la lista `ht(1, ht(2, ht(3, ht(4, nul))))`.

Unir dos listas I

1. Queremos definir una función en ML que una dos listas, es decir recibe las listas `ht(1, ht(2, nul))` y `ht(3, ht(4, nul))` y nos debe regresar la lista `ht(1, ht(2, ht(3, ht(4, nul))))`.
2. Como el tipo genérico `'a Milist` está definido de manera recursiva, debemos definir la función recursiva `uneList` para unir las dos listas.

Unir dos listas I

1. Queremos definir una función en ML que una dos listas, es decir recibe las listas `ht(1, ht(2, nul))` y `ht(3, ht(4, nul))` y nos debe regresar la lista `ht(1, ht(2, ht(3, ht(4, nul))))`.
2. Como el tipo genérico `'a Milist` está definido de manera recursiva, debemos definir la función recursiva `uneList` para unir las dos listas.
3. `'a uneList = fn : 'a Milist * 'a Milist -> 'a Milist`

Unir dos listas I

1. Queremos definir una función en ML que una dos listas, es decir recibe las listas `ht(1, ht(2, nul))` y `ht(3, ht(4, nul))` y nos debe regresar la lista `ht(1, ht(2, ht(3, ht(4, nul))))`.
2. Como el tipo genérico `'a Milist` está definido de manera recursiva, debemos definir la función recursiva `uneList` para unir las dos listas.
3. `'a uneList = fn : 'a Milist * 'a Milist -> 'a Milist`
4. De manera general podemos decir que una función definida sobre cualquier tipo debe contemplar qué hacer con cada uno de los constructores que definen a su tipo.

Unir dos listas II

```
nul ht(3, ht(4, nul))
ht(3, ht(4, nul))
```

Unir dos listas II

```
ht(1, ht(2, nul)) ht(3, ht(4, nul))
ht(1,
        ht(2, nul) ht(3, ht(4, nul))
        ht(2,
                nul ht(3, ht(4, nul))
                ht(3, ht(4, nul))
        ht(2, ht(3, ht(4, nul)))
ht(1, ht(2, ht(3, ht(4, nul))))
```

Unir dos listas III

En el caso concreto de `'a Milist` tenemos el siguiente esquema de función:

Unir dos listas III

En el caso concreto de `'a Milist` tenemos el siguiente esquema de función:

```
fun anyfun(nul ...
|   anyfun(ht(h, t) ...
```

Unir dos listas III

En el caso concreto de `'a Milist` tenemos el siguiente esquema de función:

```
fun anyfun(nul ...
|   anyfun(ht(h, t) ...

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

Unir dos listas III

En el caso concreto de `'a Milist` tenemos el siguiente esquema de función:

```
fun anyfun(nul ...
|   anyfun(ht(h, t) ...

fun uneList(nul, any) = any
|   uneList(ht(h, t), any) =

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

Unir dos listas III

En el caso concreto de `'a Milist` tenemos el siguiente esquema de función:

```
fun anyfun(nul ...
|   anyfun(ht(h, t) ...

fun uneList(nul, any) = any
|   uneList(ht(h, t), any) =

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

```
- fun uneList(nul, any) = any
|     uneList(ht(h, t), any) = ht(h, uneList(t, any));
> val 'a uneList = fn : 'a Milist * 'a Milist -> 'a Milist
-
```

Ejercicios

1. Diga que hace la siguiente función.

```
fun sabe(x, nul) = false
|   sabe(x, ht(h, t)) = if x = h
                        then true
                        else sabe(x, t);
```

2. Construya una función que elimine la primera aparición en una lista de un elemento dado.

```
- elimina1(2, ht(3, ht(2, ht(4, ht(2, nul)))));
> val it = ht(3, ht(4, ht(2, nul))) : int Milist
-
```

3. Construya una función que elimine todas las apariciones en una lista de un elemento dado.

```
- eliminat(2, ht(3, ht(2, ht(4, ht(2, nul)))));
> val it = ht(3, ht(4, nul)) : int Milist
-
```

4. Construya una función que invierta los elementos de una lista.

```
- inv(ht(1, ht(2, ht(3, ht(4, nul)))));
> val it = ht(4, ht(3, ht(2, ht(1, nul)))) : int Milist
-
```