Standard ML

Daniel Delgado, *Estudiante, ITCR,* Wilbert Gonzales, *Estudiante, ITCR,*
Anthony Leandro, *Estudiante, ITCR,* and Bryan Mena, *Estudiante, ITCR*

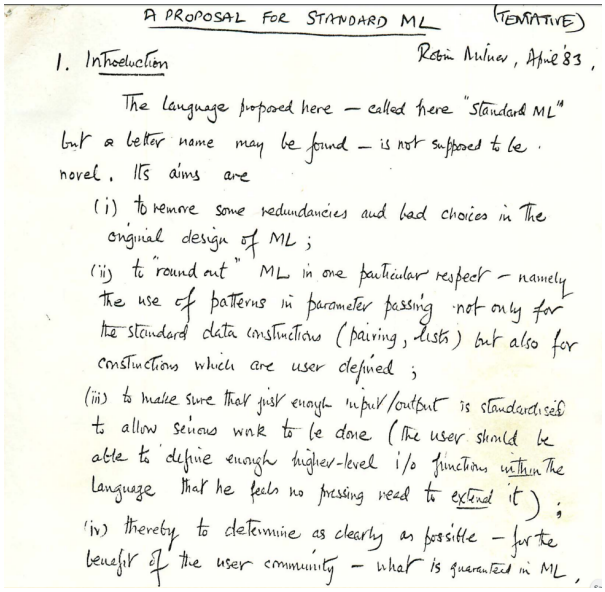
Resumen—El presente documento introduce el Standard Meta Language. En la primera sección se explica como inició el lenguaje. En la segunda sección se definirán varios tipos de datos básicos.

Es importante aclarar que el propósito de este trabajo no es ser un manual del lenguaje, para esto puede consultarse la bibliografía más detalladamente.

1. Datos Históricos

Standard Meta Language (SML), es un lenguaje de propósito general, modular, funcional con comprobación de tipo en tiempo de compilación e inferencia de tipos. SML es popular entre escritores de compiladores e investigadores de lenguajes de programación, así como en la demostración automática de teoremas.

Las primeras reuniones de diseño fueron llevadas a cabo en 1983, 1984 y 1985. A inicios de abril, en Edimburgo, se realizó la reunión inicial con el fin de discutir el primer borrador, realizado por Robin Milner, sobre la propuesta de un lenguaje nuevo, el cual incorporaba ideas de LCF/ML, VAX ML y Hope. Incluyendo a los participantes en físico y de manera virtual, eran 17 los involucrados, algunos de los asistentes fueron Rod Burstall, Luca Cardelli, Kevin Mitchell, entre otros.



A PROPOSAL FOR STANDARD ML (TENTATIVE)

Robin Milner, April '83

1. Introduction

The language proposed here — called here "Standard ML" but a better name may be found — is not supposed to be novel. Its aims are

(i) to remove some redundancies and bad choices in the original design of ML;

(ii) to "round out" ML in one particular respect — namely the use of patterns in parameter passing not only for the standard data constructions (pairing, lists) but also for constructions which are user defined;

(iii) to make sure that just enough input/output is standardised to allow serious work to be done (the user should be able to define enough higher-level i/o functions within the language that he feels no pressing need to extend it);

(iv) thereby to determine as clearly as possible — for the benefit of the user community — what is guaranteed in ML,

Figura 1: Borrador de Robin Milner

Desde 1993 hasta el 2000 se efectuaron una serie de reuniones con el objetivo de definir la "siguiente generación"de ML. Sin embargo no se alcanzó un consenso, principalmente por el desacuerdo sobre la idea de agregar o no propiedades de Orientación a Objetos al lenguaje.

2. Guía de Estilo

2.1. Identificadores

Tipo	Ejemplo	Uso
Simbólico	** ++	Operaciones especiales
Minúscula	x y z	Valores locales
Todo en minúscula	foo_bar	Valores locales
Mayúscula	FooBar	Constructores de tipos, de excepciones, estructuras, functors
Todo en mayúscula	FOO_BAR	Tipo de una estructura

2.1.1. *Convenciones*

Cuadro 1: Ejemplo convenciones

```
BIEN:
x : 'a
xs : 'a list
xss : 'a list list

MAL:
l : 'a list
list : 'a list
```

2.2. Formato

2.2.1. *Paréntesis*

Cuadro 2: Convención paréntesis

```
BIEN:
(x + y)

MAL:
( x + y )
```

2.2.2. Operadores

Cuadro 3: Convención operadores

BIEN:
```
x + y
```

MAL:
```
x+y
```

2.2.3. Espacios en blanco

Cuadro 4: Convención espacios en blanco

BIEN:
```
[x, y, z]
```

MAL:
```
[x,y,z]
[x , y , z]
```

2.2.4. Identación

Por defecto el tamaño de la identación es de dos espacios.

Cuadro 5: Convención identación

BIEN:
```
if b then e1 else e2

if b1 then e1
else if b2 then e2
else e3

if b1 then
  e1
else if b2 then
  e2
else
  e3
```

MAL:
```
if b then e1
  else if b2 then e2
    else e3
```

2.3. Recomendaciones generales

- Usar nombres cortos para valores locales.
- Es preferible utilizar nombres largos que nombres poco representativos.

Cuadro 6: Recomendación nombres

BIEN:
```
fun remueve_duplicados xs = ...
```

MAL:
```
(*esta funci\'on elimina
duplicados de una lista*)
fun procesa_lista xs = ...
```

- Evitar comentarios sin sentido.

Cuadro 7: Recomendación comentarios

MAL:
```
(*esta funci\'on elimina
duplicados de una lista*)
fun remueve_duplicados xs = ...
```

- Nunca mezclar convenciones en definición de nombres.
- Nunca mezclar idiomas en el código.

3. Tipos de Datos

Existen diferentes tipos básicos pre-definidos en SML. Por ejemplo el tipo de valores lógicos, llamado *bool*. Esto tiene exactamente dos valores, verdadero o falso. Otro tipo simple es el *string*. Los *strings* se colocan entre comillas dobles (como en "Hola") y se unen con " ^ " (el smbolo del cursor o flecha hacia arriba). Como era de esperar, la expresin "Hola" ^ "mundo!" se evalúa como "Hola mundo!". También hay un tipo *char* para caracteres individuales. Una constante *char* es una constante *string* de longitud uno precedida por un símbolo de hash. Así, #"a" es la letra a. Se puede acceder a los caracteres de ocho bits mediante su código ASCII.

Los tipos numricos en SML son los enteros de tipo *int*, los números reales de tipo *real* y enteros sin signo (o palabras) de tipo *word*. Además de esto, una implementación del lenguaje puede proporcionar otros tipos numéricos en una biblioteca, por ejemplo, enteros de precisión arbitraria, reales de doble precisión o incluso bytes (enteros sin signo de 8 bits). Los números enteros se pueden representar en notación decimal o hexadecimal, de modo que 255 y 0xff representan el número entero 255.

Los tipos numéricos mencionados son tipos separados y deben usarse funciones para convertir un entero a un real o un word a un byte. No hay conversión implícita entre tipos como en otros lenguajes de programación.

3.1. Estructuras Básicas de datos

- Numéricos
  int (integer), como un ∼7 o 255. Nótese que una virgulilla '∼' es usada para números negativos.
  real (número de punto flotante), como un ∼7.8 o 9.4.
  SML no hace conversiones numéricas entre int y real automáticamente. Por lo tanto una expresión como 9 + 3.66 es inválida. Debe ser escrito como 9.0 + 3.66, o real(9) + 3.66 (usando la función real para convertir 9 a 9.0).
- Caracteres
  string (cadena de caracteres), como "esto es una cadena" o "". Nótese que el segundo es una cadena vacía, contiene cero caracteres.
  char (un caracter), como #"y".
- Lógico
  bool (valor Booleano), este puede ser verdadero o falso.

3.2. SML Basis Library

La biblioteca *SML Basis* es el estándar para la revisión de SML en 1997. Esta biblioteca provee interfaces y operaciones para tipos básicos, como integers y strings, soporte para entrada y salida, soporte para tipos de datos estándar, como lo son *options* y las listas. Algunas de las funciones de esta biblioteca son:

- Constantes matemáticas básicas como raíz cuadrada, funciones trigonométricas, hiperbólicas, exponenciales y logarítmicas.
- Tener en un *real* valores positivos o negativos infinitos.
- Conversión de valores booleanos a *string*.
- Representación e intervalos de tiempo.
- Conversiones entre valores numéricos.
- Búsquedas en cadenas de*char*.
- Operaciones sobre *strings*.
- Operaciones sobre listas.
- Obtener valor absoluto.
- Longitud de un array.

3.3. Opciones

El tipo de dato *option* puede utilizarse cuando es posible que algo no tenga un valor válido. Por ejemplo, una división entre cero, sin tener que recurrir al manejo de excepciones.

Cuadro 8: Ejemplo División entre 0

```
fun dividir x y = if y == 0
then NONE else SOME (x / y)
```

3.4. Listas

Las listas en SML son un tipo de dato recursivo y polimórfico; sumamente importantes y comunes en la programación funcional. La estructura de una lista es la siguiente: cabeza (head) se encuentra en la primer posición (de izquierda a derecha) y la cola (tail) corresponde al resto de elementos. Cada elemento de la lista tiene un índice, iniciando con el valor 0 para la cabeza.

Cuadro 9: Ejemplo Listas

```
datatype 'a list = nil
| :: of 'a * 'a list
```

'::' es un operador infijo, por ejemplo, *3 :: 4 :: 5 :: nil* es una lista de tres enteros (int). También es posible declarar listas de la siguiente forma: *[3, 4, 5]*.

3.5. Tuplas

Los tipos de datos, incluidos los tipos bsicos anteriores, se pueden combinar de diferentes maneras. Una forma de hacer esto es en una tupla, que es un conjunto ordenado de valores; por ejemplo, la expresin (1, 2) es del tipo int * int, y ('foo', false) es del tipo string * bool. También hay una 0-tupla, (), cuyo tipo se denomina *unit*. Las tuplas pueden ser anidadas, y (a diferencia de algunos formalismos matemáticos), (1, 2, 3) es distinto de ((1, 2), 3) y de (1, (2, 3)). El primero es de tipo int * int * int; los otros dos son de tipos (int * int) * int e int * (int * int), respectivamente. Ejemplos de declaraciones de tuplas:

Cuadro 10: Ejemplo Listas

```
val alien1 = ("Groot", 1586, 3.14159)

val gustos =
[ ("Gamora", "helados"),
("Drax",   "perritos"),
("Rocket",   "cerrezas") ]

val combinado =
[ ("Gamora", 27),
("Drax",   78),
("Rocket",   15) ]

val articulos =
(["helado", "perritos",
 "chocolate"], ["higado",
 "alquiler" ])
```

3.6. Tokens

Un programa escrito en SML consiste de una secuencia de *tokens*, algunos de los más comunes son:

Tipo de token	Ejemplos
Identificadores alfanuméricos	x, mod, 'a
Identificadores simbólicos	+, -, *, /
Constantes especiales	2, 5.6, "string", #'c'
Palabras clave	val, =, (,)

4. FUNCIONES

En SML una función acepta un valor y normalmente devuelve otro valor. Como puede verse en el cuadro 4 la función factorial es de tipo *int ->int*, esto significa que la función recibe un valor de tipo **int** y retorna un valor de tipo **int**. El *binding* para la función debe hacerse como el ejemplo.

Cuadro 11: Ejemplo Factorial

```
fun factorial n =  if n < 1
then 1  else n * factorial (n - 1)
```

Incluso si una función no retorna un valor en tiempo de ejecución (por una excepción o si entra a un loop infinito) tiene un tipo de retorno estático en tiempo de compilación. Es importante recordar que las funciones en SML son valores y todos los valores pueden estar enlazados a nombres, por lo tanto funciones pueden estar enlazadas a nombres.

Cuando en el cuerpo no se provee de suficiente información para determinar el tipo exacto del argumento de retorno, puede definirse el tipo explícitamente de la siguiente manera:

Cuadro 12: Ejemplo Factorial Tipo Explícito

```
fun factorial (n : int) : int = if n<1
then 1  else n * factorial (n - 1)
```

4.1. Comprobación de tipos

Los tipos de las llamadas a funciones son comprobadas de la manera más obvia. El argumento actual debe coincidir con el argumento formal, cuando no es así se genera un error.

4.2. Funciones sin valores de retorno o argumentos

Lenguajes como Pascal resuelven este problema dividiendo el universo de abstracciones de control en dos tipos: funciones, que devuelven valores y procedimientos, que no. Los lenguajes como C solucionan este problema al tener funciones *void* que no devuelven nada. SML utiliza un enfoque similar, pero no igual a este ltimo: utiliza el tipo *unit*, que tiene un valor, escrito (): *printHi()*

4.3. Funciones recursivas

Las funciones recursivas son aquellas que utilizan algún tipo de iteración para obtener los resultados. En los lenguajes imperativos la iteración se logra utilizando ciclos *while* y ciclos *for*, en SML se hace mediante la recursión.

Una función definida por recursión es una que calcula el resultado de una llamada "llamándose a sí misma". Para hacer esto la función debe tener un nombre con el cual se hará referencia a la misma. Se logra usando un *binding* de valor recursivo.

Bindings de valor recursivo tienen casi la misma forma que los ordinarios, excepto que el *binding* es calificado con el adjetivo rec", como en el ejemplo siguiente:

Cuadro 13: Ejemplo Factorial Recursivo

```
val rec factorial : int->int = fn 0=>1
| n:int => n * factorial (n-1)
```

Usando la notación de *fun* puede escribirse más claramente y conciso así:

Cuadro 14: Ejemplo Factorial Recursivo Fun

```
fun factorial 0 = 1
|factorial (n:int) = n*factorial (n-1)
```

En ciertas ocasiones es útil definir dos funciones simultáneamente, donde cada una de las cuales se llama a sí misma y/o la otra función para calcular el resultado. Este tipo de funciones se les llama *mutuamente recursivas*, esto se hace de la siguiente manera:

Cuadro 15: Ejemplo Funciones Mutuamente Recursivas

```
fun par 0 = true
| par n = impar (n-1)
and impar 0 = false
| impar n = par (n-1)
```

4.4. Ramificación (branching)

Los lenguajes imperativos expresan ramificacin a través de declaraciones condicionales, sin embargo lenguajes funcionales como SML, expresan ramificacin principalmente a través de expresiones condicionales.

4.5. Expresiones if

Los condicionales if en SML tienen la siguiente sintaxis: *if boolExpr then expr1 else expr2*

Esto tiene la semántica similar al operador :? de C:

1. Primero, boolExpr es evaluado.
2. Si la expresión es verdadera, entonces expr1 es evaluada y retornada.
3. Si la expresión es falsa, entonces la expr2 es evaluada y retornada.

4.6. Secuenciadores (sequencing)

Las secuencias de expresiones en SML son escritas separadas por punto y coma entre paréntesis.

Secuencias de expresión tienen la siguiente semántica:

1. Evaluar cada expresión en orden de izquierda a derecha.
2. Devuelven la última expresin evaluada como el valor de toda la expresión.

Todos los resultados aparte de la última expresión son descartados. Las secuencias de expresiones son primordialmente útiles para las expresiones de efectos secundarios como *print*:

Cuadro 16: Ejemplo Secuencias de Expresiones

```
- val x = (print "Hola\n"; 7)

Output:
Hola
val x = 7 : int
```

4.7. Comparación de patrones

La expresión if básicamente provee una manera de igualar un valor booleano contra un verdadero o falso. Otra manera de escribir esto en SML es:

Cuadro 17: Ejemplo Secuencias de Expresiones

```
case boolExpr of
true => expr1
| false => expr2
```

El constructo case toma un valor e intenta igualarlo a uno o más patrones, en este caso, los dos patrones constantes booleanos, falso y verdadero. Si un patrón es igual, entonces su correspondiente función es evaluada y retornada como el valor de la expresión completa. Los patrones son probados de izquierda a derecha.

Los patrones no están restringidos a ser usados en declaraciones *case*. Podrían aparecer donde cualquier *binding* aparezca, incluyendo declaraciones *val* y argumentos de funciones.

4.8. Polimorfismo

En muchos lenguajes de programacin (por ejemplo, Java) encontramos que terminamos reescribiendo el mismo código una y otra vez para que funcione con diferentes tipos. SML no tiene este problema, pero tenemos que introducir nuevas características para mostrar cómo evitarlo. Supongamos que queremos escribir una función que intercambia la posición de valores en un par ordenado:

Cuadro 18: Ejemplo Polimorfismo 1

```
fun swapInt(x: int, y: int):
int*int = (y,x)

fun swapReal(x: real, y: real):
real*real = (y,x)

fun swapString(x: string, y: string):
string*string = (y,x)
```

Esto es tedioso, porque estamos escribiendo exactamente el mismo código cada vez. ¿Qué sucede si los elementos de dos pares tienen diferentes tipos?

Cuadro 19: Ejemplo Polimorfismo 2

```
fun swapIntReal(x: int, y: real):
real*int = (y,x)

fun swapRealInt(x: real, y: int):
int*real = (y,x)
```

La mejor manera de hacer esto es:

Cuadro 20: Ejemplo Polimorfismo correcto

```
- fun swap(x: 'a, y: 'b):
  'b * 'a = (y,x)

val swap = fn :
  'a * 'b -> 'b * 'a
```

En lugar de escribir tipos explícitos para x e y, se escriben las variables de tipo 'a y b'. El tipo de swap es 'a*'b -¿'b*' a. Lo que esto significa es que podemos usar *swap* como si tuviera cualquier tipo que se pudiese obtener consistentemente reemplazando 'a' y 'b' en su tipo con un tipo para 'a y un tipo para' b. Puede usarse el nuevo *swap* en lugar de todas las viejas definiciones:

Cuadro 21: Ejemplo Polimorfismo con definiciones antiguas

```
(*swap:(int*int)->(int*int)*)
swap(1,2);

(*swap:(real*real)->(real*real)*)
swap(3.14,2.17);

(*swap:(string*string)->(string*string)*)
swap("foo","bar");

(*swap:(string*real)->(real*string)*)
swap("foo",3.14);
```

Esta capacidad de usar *swap* como si tuviera muchos tipos diferentes se conoce como polimorfismo, del griego para "muchas formas". Si pensamos que *swap* tiene una "forma"que su tipo define, entonces *swap* puede tener muchas formas: es polimórfico. Obsérvese que el requisito de que las variables de tipo sean sustituidas consistentemente significa que algunos tipos están descartados; por ejemplo, es imposible usar swap en el tipo (int * real) -¿(string * int), porque ese tipo sustituiría consistentemente por la variable de tipo 'a pero no por 'b.

Los programadores de SML suelen leer los tipos 'a' y 'b' como .ªlfaz "beta". Esto es más fácil que decir çomillas simples a". De hecho, una variable de tipo puede ser cualquier identificador precedido por una comilla simple; por ejemplo, el valor *'key* y *'valor* también son variables legales. El compilador SML necesita tener estos identificadores precedidos por una comilla simple para que sepa que est'a viendo una variable de tipo.

Es importante tener en cuenta que el *swap* no utiliza sus argumentos x o y de manera interesante. Los trata como si fueran cajas negras. Cuando el verificador de tipo SML está comprobando la definición de *swap*, todo lo que sabe es que x es de algún tipo arbitrario 'a. No permite realizar ninguna operación en x que no pueda realizarse en un tipo arbitrario. Esto significa que el código está garantizado para trabajar para cualquier x e y.

5. Características

- SML cuenta con un sistema de tipado estático particularmente fuerte. A diferencia de muchos lenguajes, no tiene subtipos (relaciones 'is-a') o implícitos entre tipos.
- El término *strong typing* se utiliza en una amplia variedad de maneras diferentes; no obstante, es justo decir que SML proporciona tipificación fuerte por casi todas las definiciones.
- Cada expresión en un programa SML tiene un tipo específico en tiempo de compilación y su tipo en tiempo de ejecución nunca se contrapone a esto.
- Todas las conversiones de tipos son explícitas (utilizando funciones como *real*, que acepta un número entero y devuelve un número real equivalente), y toman la forma de traducciones significativas en lugar de meras reinterpretaciones de bits en bruto.
- La mayoría de los lenguajes no se adhieren estrictamente a uno solo de los enfoques de tipado, sino más bien, utilizan elementos de más de uno. El sistema de tipos de SML, sin embargo, usa la tipificación estática casi exclusivamente.
- Un programa mal tipificado ni siquiera compilará. En la medida en que SML admite el tipo dinámico, es dentro del marco de tipificación estática.

6. Ventajas

- Permite a los programadores definir tipos de datos abstractos.
- Los programadores de SML consideran el estilo de tipado de SML un aspecto positivo, ya que permite que muchos errores de programación sean capturados en tiempo de compilación que un lenguaje de tipo dinámico solo captura en tiempo de ejecución.
- En SML/NJ se puede construir y manejar estructuras de datos de C arbitrarias en código SML.
- En SML/NJ Puede cargar bibliotecas de C y encontrar funciones/variables en tiempo de ejecución.
- En SML/NJ Puede administrar memoria de estructuras de datos de C. Si se sabe que algo no se usa, puede liberarse.

7. Desventajas

- Si el programador se encuentra acostumbrado a los demás lenguajes donde el tipado es distinto, encontrarse con SML puede significar un mayor tiempo en el aprendizaje de este.
- Los tipos, en comparación con otros lenguajes, son un poco más complejos de comprender.

8. CÓDIGO DE EJEMPLO

El programa "hello.sml"

Cuadro 22: Ejemplo Hello World

```
print "Hello, World!\n";
```

Puede compilarse con MLton:

Cuadro 23: Compilar Hello World

```
$ mlton hello.sml
```

Y ejecutarse:

Cuadro 24: Ejecución Hello World

```
./hello

Hello, World!
```

REFERENCIAS

[1] CSE341 Lecture Notes 3: Functions and patterns in ML. (s.f.). Recuperado 3 de octubre de 2017, a partir de https://courses.cs.washington.edu/courses/cse341/04wi/lectures/03-ml-functions.html

[2] Learn Standard ML in Y Minutes. (s.f.). Recuperado 18 de septiembre de 2017, a partir de https://learnxinyminutes.com/docs/standard-ml/

[3] Lecture 4: Polymorphism and Parameterized Types. (s.f.). Recuperado 6 de octubre de 2017, a partir de http://www.cs.cornell.edu/courses/cs312/2008sp/lectures/lec04.html

[4] ML2015-talk.pdf. (s.f.). Recuperado a partir de http://sml-family.org/history/ML2015-talk.pdf

[5] Recursive Functions. (s.f.). Recuperado 3 de octubre de 2017, a partir de https://www.cs.cmu.edu/ rwh/introsml/core/recfns.htm

[6] sml-intro.pdf. (s.f.). Recuperado a partir de https://www.cs.cmu.edu/afs/cs/academic/class/15814-f03/www/sml-intro.pdf