Universidad Nacional de Colombia

Facultad de Ingeniería
Departamento de Sistemas e Industrial

Tutorial Mercury

Stevan Valbuena Gaviria
Samuel Salgado Rivera
Diego Bulla Poveda

Lenguajes de programación
2023-1

Introducción

Mercury es un lenguaje de programación lógica / funcional que combina la claridad y la expresividad de la programación declarativa con funciones avanzadas de análisis estático y detección de errores. Como lenguaje lógico, está basado en el cálculo de predicados de primer orden y utiliza un sistema de inferencia automático para deducir conclusiones a partir de hechos y reglas declarados.

Dentro de sus características, se destacan:

- Basado en prolog (recomendamos el tutorial disponible aquí)
- Soporta modos
- Tiene un fuerte sistema de determinismo
- Tiene garbage collector
- Es modularizado
- Su compilador facilita el análisis estático y la optimización de código
- Variedad de lenguajes de destino
- Compila a código de máquina
- Contiene un sistema de depuración avanzada
- Permite generar automáticamente documentación a partir de anotaciones dentro del código

Instalación

Unix *(recomendado):*

1. Se requiere instalar GNU C (gcc) y GNU Make

```
sudo apt update

sudo apt install build-essential
```

2. Descargar la clave GPG del autor del software y autorizar al sistema operativo para descargar cualquier software publicado por el autor.

```
sudo apt install wget ca-certificates

cd /tmp

wget https://paul.bone.id.au/paul.asc

sudo cp paul.asc /etc/apt/trusted.gpg.d/paulbone.asc
```

3. Se debe agregar el repositorio de software al sistema operativo.

```
sudo nano /etc/apt/sources.list

# Dentro del archivo, agregar las siguientes líneas

deb http://dl.mercurylang.org/deb/ DISTRO main

deb-src http://dl.mercurylang.org/deb/ DISTRO main

# Donde “DISTRO” es el nombre de versión del sistema operativo (sid, bookworm, bullseye, disco, focal, jammy)
```

4. Instalar el conjunto de paquetes recomendado.

```
sudo apt install mercury-recommended
```

NOTA: Herramienta para trabajar el lenguaje online: https://glot.io/new/mercury

Compilación

Inicialmente se crea un archivo nombre.m (nombre es definido por el programador) dónde irá el código del programa. Para compilarlo en la consola basta con escribir: “mmc nombre.m”. mmc hace referencia al compilador de Mercury.

Ejemplo: hello.m

```
:- module hello.

:- interface.

:- import_module io.

:- pred main(io::di, io::uo) is det.

:- implementation.

main(!IO) :-

     io.write_string("Hello, world!\n", !IO).
```

```
# Compilar
mmc --make hello
# Ejecutar
./hello
```

Estructura del código

Ahora se explicará la estructura del ejemplo anterior de “hola mundo” para ver como está compuesto el código

Modulo llamado hello, en un archivo llamado hello.m

```
     :- modulo hello.
```

Todo lo que va después de interface será visto por quienes importen el módulo que estamos escribiendo.

```
:- interface.
```

Importa módulos que serán utilizados, pueden ser de la librería estándar de Mercury o creados por el programador.

```
:- import_module io.
```

Se crea un predicado en este caso determinista main de dos argumentos, uno en modo entrada y otro en modo salida.

```
:- pred main(io::di, io::uo) is det
```

Todo lo que va después de implementación es de tipo privado y solo está visible para el módulo actual.

```
:- implementation.
```

Se crea una cláusula main con cabeza y cuerpo separados por el símbolo :- donde el cuerpo se compone de un solo “goal”.

```
main(!IO) :-
     io.write_string("Hello, World!\n", !IO).
```

Tipos

Mercury contiene los tipos clásicos int, float, char, tuple, etc. Sin embargo el usuario también puede definir nuevos tipos fácilmente utilizando la palabra reservada type, a continuación se mostraran algunos junto con algunos ejemplos.

- **int**
  Sintácticamente, un int es una secuencia de dígitos, opcionalmente precedida por un signo menos.

Ejemplos

decimal -123, 0, 42
hexadecimal -0x7B, 0x0, 0x2a
octal -0o173, 0o0, 0o52
binary -0b1111011, 0b0, 0b101010.

- **float**
  Sintácticamente, un float es un número de punto flotante decimal (se requiere el punto decimal), precedido opcionalmente por un signo menos, opcionalmente seguido de un exponente entero.

  Ejemplos

  1.414
  1414e-3
  .1414e1
  0.01414e2

- **String**
  Una constante de cadena es una secuencia de caracteres entre comillas dobles.

  Ejemplos

  " "
  "Hello, World!\n"
  "\“Lawks!\” I declared."

- **Char**
  Sintácticamente, son caracteres individuales (o caracteres de escape) entre comillas simples.

  Ejemplos

’A’
’\x41\’
’\101\’
’\’’
’\n’

NOTA: El lenguaje también contiene el siguiente conjunto de caracteres de escape:

\” Comilla doble
\’ Comilla simple
\b Backspace
\n Nueva línea
\\ Backslash
\t Tabular
\v Tabulación vertical

- **Tuplas**
  Sintácticamente, un tipo de tupla es una coma secuencia separada de nombres de tipos encerrados entre llaves, mientras que un valor de tupla es una secuencia de valores separados por comas encerrados entre llaves.

  Ejemplos:

  -> {111, 'b'} es un valor de tipo {int, char}
  -> {1.2, 3, "456"} es un valor de tipo {float, int, string}
  -> {"un", {"pequeño", "artificial"}} es un valor de tipo {cadena, {cadena, cadena}}.

- **Listas**
  Sintácticamente, una lista es una secuencia de valores del mismo tipo separados por comas entre paréntesis.

Ejemplos:

-> [] indica la lista vacía, independientemente del tipo de lista
-> [1, 2, 3] es un valor de tipo list(int)
-> ['a', 'b', 'c', 'd'] es un valor de tipo lista(caracter)
-> [[1], [2, 3], [4]] es un valor de tipo list(list(int)).

- **Uniones discriminadas**
  Las uniones discriminadas permiten la definición de nuevos tipos estructurados.

  Ejemplo:

  ```
  :- type playing_card ---> card(rank, suit) ; joker.

  :- type rank ---> ace          ; two        ; three          ; four
                  ; five         ; six        ; seven          ; eight
                  ; nine         ; ten        ; jack       ; queen
                  ;    king.

  :- type suit ---> clubs ; diamonds ; hearts ; spades.
  ```

- **Equivalencias**
  La legibilidad a menudo se mejora dando nombres simples a tipos complejos o mediante el uso de nombres más significativos para usos específicos de tipos generales:

  Ejemplo:

  ```
  :- type height == float. %In metres
  :- type radius == float. %In metres
  :- type volume == float. %In cubic metres

  :- func volume_of_cylinder(height, radius) = volume.
  :- func volume_of_sphere(radius) = volume.
  ```

Modos

Los predicados en Mercury son similares a los procedimientos en otros lenguajes, no tienen un valor de retorno por sí solos. En cambio tienen valores de verdad de acuerdo con las entradas y salidas que reciben como argumento.

La declaración de predicados tiene la siguiente sintaxis:

```
:- pred nombre_predicado (arg1 :: modo, arg2 :: modo, ...)
determinismo
```

Inician con la palabra reservada pred seguido del nombre del predicado y entre paréntesis sus argumentos, que pueden ser tipos o variables seguidos cada uno del modo en el que serán interpretados. Luego, se escribe el determinismo del predicado que puede ser opcional. Si no se escribe, por defecto es determinista.

La definición de predicados tiene la siguiente sintaxis:

```
nombre_predicado (arg1, arg2, ... ) :- cuerpo
```

Se compone de dos partes, cabeza y cuerpo. En la cabeza se determina el nombre del predicado y los parámetros con los que va a trabajar. En el cuerpo se especifica cómo va a trabajar.

Ejemplo:

```
:- pred phone(string::in, int::out) is semidet.
phone("Ian", 66532).
phone("Julien", 66532).
phone("Peter", 66540).
phone("Ralph", 66532).
phone("Zoltan", 66514).

#El estilo de declaración pred que se usa aquí se denomina
declaración de pred-modo.
```

La siguiente declaración

```
:- pred phone(string::in, int::out) is semidet.
```

se puede expresar como:

```
:- pred phone(string, int).
:- mode phone(in, out) is semidet.
```

La declaración pred nos dice los tipos de argumentos, la declaración de modo nos dice los modos del argumento (in o out) y la categoría de determinismo correspondiente (semidet).

1. ¿phone("Harald", HaraldsNum)? -> Sin respuesta
2. ¿phone("Ralph", RalphsNum)? -> 66532

¿Y si queremos buscar el nombre asociado en base al número telefónico?

En este caso es necesario más de una declaración de modo

```
:- pred phone(string, int).
:- mode phone(in, out) is semidet.
:- mode phone(out, in) is nondet.
```

1. ¿(Persona, 12345)? -> Sin respuesta
2. ¿(Persona, 66532)? -> Ian, Julien, Ralph

Categorías de determinismo

Una categoría de determinismo nos dice si un procedimiento en particular puede fallar y si puede tener más de una solución:

Categoría	# Soluciones
det	1
semidet	<= 1
multi	>= 1
nondet	>= 0
failure	0