Grob: Programación funcional reactiva para robots con bajas capacidades de cómputo

Guillermo Pacheco

2 de mayo de 2015

Resumen

El proyecto consiste en la creación de un lenguaje de programación para robots, cuyas capacidades de cómputo son limitadas. Para esto se escogió el paradigma de Programación Funcional Reactiva (*FRP*) el cuál permite expresar naturalmente reacciones a valores que varían en función del tiempo.

El objetivo es utilizarlo con fines educativos, por lo tanto debe ser simple y fácil de usar por usuarios inexpertos y no familiarizados con la electrónica y la informática.

Para resolver el problema, se dividió en dos etapas.

La primera consiste en la implementación de un compilador que traduce el lenguaje cuyo nivel es alto a un lenguaje de bajo nivel (*Bytecode*) más simple e interpretable.

La segunda etapa consiste en implementar una máquina virtual, que sea capaz de interpretar el lenguaje de bajo nivel. Por cada plataforma objetivo, es posible realizar una implementación de la máquina, lo que permite ejecutar los mismos programas en alto nivel, en diferentes plataformas.

El lenguaje se implementará como un DSL embebido en Haskell.

El diseño de la máquina consiste de un núcleo común capaz de interpretar instrucciones, y módulos bien definidos de entrada/salida los cuáles varían de una plataforma a otra. Ésto permite mayor portabilidad y extensibilidad.

Debe ser posible ejecutar programas en dicho lenguaje dentro de plataformas de hardware reducido. Considerando ésto el lenguaje de programación elegido para la implementación de la máquina virtual es C/C++.

De ésta forma se implementación de un lenguaje reactivo con las características deseadas, y una implementación modelo de una máquina virtual que permite su ejecución en una arquitectura objetivo deseada. La misma es fácil de mantener, portable y cuenta con suficiente flexibilidad para ser extendida.

Índice general

Índice de figuras

Índice de cuadros

Capítulo 1

Introducción

Este documento desarrolla el proceso de construcción de herramientas que permitan programar robots utilizando el paradigma de programación funcional reactiva.

Para ello se define el lenguaje Grob de alto nivel, que permite expresar los comportamientos de un robot y sus interacciones. El lenguaje permite expresar los mismos en base a *Eventos* y relaciones entre ellos, que permiten capturar la naturaleza reactiva del dominio.

Éste lenguaje fue construído con fines académicos y de investigación, uno de los objetivos es que sea simple de utilizar e intuitivo.

Otro objetivo planteado es que debe ser posible programar con él sistemas robóticos embebidos en plataformas de hardware de baja capacidad de cómputo, y en lo posible bajo costo.

En los siguientes capítulos se introduce el concepto de programación funcional reactiva, junto con sus variantes y algunos ejemplos. Luego de decidir cuál de éstas variantes es más útil en nuestro dominio, se muestra la definición del lenguaje Grob y cuál es el proceso desde que se escribe un programa hasta que es ejecutado dentro de una plataforma.

Para cumplir con el objetivo de que la implementación sea portable, se separó la misma en dos fases. Los programas son compilados a código de menor abstracción, el cuál es ejecutado por una máquina virtual que definiremos. La misma puede ser portada a distintas plataformas.

Para finalizar se trata un caso de estudio completo, en el que se implementa un desafío robótico utilizado en la competencia SumoUY.

Capítulo 2

Programación Funcional Reactiva

2.1. Programación Funcional Reactiva

Tradicionalmente los programas son formados a partir de una secuencia de acciones imperativas. Los programas reactivos suelen formarse por eventos y código iterativo que se corre cuando un evento ocurre.

Dicho código iterativo suele hacer referencia y manipular variables compartidas con diferentes rutinas. Ésto lleva a que como un valor puede ser manipulado desde diferentes lugares, puedan producirse problemas de concurrencia y algunos valores pueden quedar en un estado inconsistente.

En el paradigma FRP no existen valores compartidos, sinó que dichos valores dependientes del tiempo, tienen una representación llamada Comportamiento y la única forma de modificarlos, es a partir de como fueron definidos.

Definición 1. *Programa reactivo.*
Es aquel que interactúa con el ambiente, intercalando entradas y salidas dependientes del tiempo. Por ejemplo un reproductor de música, video juegos o controladores robóticos.

Difiere de los programas transformacionales *los cuáles toman una entrada al inicio de la ejecución y producen una salida completa al final. Por ejemplo un compilador.*

Definición 2. *Comportamientos (Behaviours).*
Un comportamiento es un valor contínuo que depende del paso del tiempo. Los comportamientos se pueden definir, combinar, pasarlos como argumentos a funciones, retornarlos. Un comportamiento puede ser un valor constante,

el tiempo mismo (un reloj), o puede formarse combinando otros comportamientos, por ejemplo secuencialmente o paralelamente.

Definición 3. *Eventos (Streams).*
Son valores discretos dependientes del tiempo, que forman una secuencia finita o infinita de ocurrencias. Cada ocurrencia está formada por el valor y el instante de tiempo.

La principal diferencia entre Comportamientos y Eventos, es que los comportamientos son valores contínuos y los eventos son discretos.

Los comportamientos representan cualquier valor en función del tiempo, por ejemplo:

- *entrada* sensor de distancia, temperatura, video
- *salida* velocidad, voltaje
- *valores* intermedios calculados

Las operaciones que se pueden realizar sobre los comportamientos incluyen:

- *Operaciones genéricas* Aritmética, integración, diferenciación
- *Operaciones específicas de un dominio* como escalar video, aplicar filtros, detección de patrones.

Los eventos pueden ser sensores específicos de un dominio, por ejemplo un botón, un click, una interrupción o mensaje asincrónico. También puede ser generado a partir de valores de un comportamiento, como ser *Temperatura alta*, *Batería baja*.

Las operaciones que se pueden realizar sobre los eventos incluyen:

- fmap, filter
- Modificar un *Comportamiento* reactivo

2.2. Ejemplo

Para entender un poco más las ideas de Comportamiento y Evento, se puede plantear el siguiente ejemplo.

En una cuenta de un banco, el saldo se puede definir como un comportamiento, el cuál solo se modifica cuando ocurre un movimiento. Un movimiento puede ser depositar dinero o extraer dinero de la cuenta. Un dato importante

a ver, es que no es posible asignar un valor sin que sea por medio de su propia definición, por lo que nadie podría realizar la asignación $saldo = 1000000$. La misma operación sería posible creando un movimiento, el cuál afectaría al saldo.

Usaremos la notación `<nombre>:: Event <tipo>` para definir fuentes de eventos. Asumo conocida la fuente de eventos *movimientos* que por cada movimiento emite su valor.

```
movimientos :: Event Number
```

Ahora puedo expresar el saldo a partir de los mismos, y construir un programa que muestre el saldo.

```
main = let saldos = (foldlE (+) SALDO_INICIAL movimientos) in
     mapE show saldos
```

En el ejemplo se ve que es imposible asignar un valor al saldo, éste se compone a partir del `SALDO_INICIAL` y la suma de los movimientos (positivos o negativos).

TODO: Mejorar esta sección explicando: Signal y SF (Signal Functions), Arrows, Behaviours y Event, Event, time leaks.

Leer y citar [1] y también este otro [2] y [3].

Capítulo 3

Diseño

En esta seccion se describe el diseño del lenguaje, su semántica, así como el diseño del lenguaje de bajo nivel, y cómo es interpretado.

3.1. Lenguaje de alto nivel

3.1.1. Sintaxis

3.1.2. Semántica

3.1.3. Ejemplo

En el siguiente ejemplo se muestra como haría para detener un robot cuando su sensor de distancia muestra un valor menor a un mínimo.

```
MINIMO = 30
INPUT_DISTANCE = 1
OUTPUT_MOTOR = 1

distanceToSpeed :: Number -> Number
distanceToSpeed n =
   if (n < MINIMO) 0 else 100

main :: IO()
main = let distance = read(INPUT_DISTANCE) in
   output OUTPUT_MOTOR (mapE (distanceToSpeed) distance)
```

Primero se definió una fuente de eventos llamada *distance*, de tipo *Event Number*.

```
let distance = read(INPUT_DISTANCE)
```

La misma emitirá un evento con la distancia en *cm* leída por un sensor en el robot.

Luego, se quiere que cuando la distancia sea menor al valor *MINIMO* dado, el robot se detenga completamente, ésto quiere decir, que la velocidad sea 0, en otro caso el mismo se mueve a velocidad 100, la cuál asumimos es una velocidad apropiada. Para ésto se crea una nueva fuente de eventos, definida a partir de *distance*, aplicandole la función `distanceToSpeed` a cada valor para obtener la velocidad.

```
(mapE (distanceToSpeed) distance)
```

Ésta expresión, también es una fuente de eventos de tipo *Event Number*.

Finalmente, se aplica la función nativa `output` cuyo primer parámetro es el *id* de la salida, en éste caso es `OUTPUT_MOTOR` y su segundo parámetro es una fuente de eventos, en éste caso son velocidades.

3.2. Lenguaje de bajo nivel

Para presentar el lenguaje, primero defino las estructuras necesarias para explicar la semántica de cada instrucción.

1. *Inputs*

   Es utilizado para almacenar todos los eventos de una aplicación. Es una cola de eventos.

   Los valores leídos en las entradas de hardware se mapean en esta lista. Por ejemplo si el hardware cuenta con un botón, y el identificador del botón es i, su valor se representará con la notación:

   $Inputs_i$

   A cada entrada se le asocia un conjunto de rutinas que deben invocarse cuando se tenga un valor disponible en la entrada. El mismo puede ser vacío si no hay rutinas esperando por su valor. Queda a criterio de quien implementa la máquina si éstos valores deben ser guardados o descartados. A éste conjunto de rutinas lo denotamos cómo:

   $Callbacks_i$

2. *Events*

3. Stack

   Es una pila global, se utiliza para ejecutar operaciones, realizar cálculos, es único y global, y los hilos de ejecución no pueden guardar valores persistentes en él.

   Usaremos el símbolo TOS para referirnos al elemento en el tope de la pila y $Stack$ para referirnos a la pila.

4. Ready

   Es una *cola* que contiene punteros a hilos de ejecución listos para ser ejecutados.

3.2.1. Instrucciones

Las siguiente tabla muestra las instrucciones de bajo nivel junto con su semántica.

- Recv inm

  $nextip = ip + 1$

  $CallBacks_{inm} := CallBacks_{inm} \cup nextip$

  $ip = 0$

- Send inm value

  Length: 2

  $Events_{inm} := Events_{inm} : \{inm, value, now\}$

- Output inm value

Capítulo 4

Implementación

En este capitulo se detalla la implementación de el compilador y la máquina virtual diseñadas para utilizar el lenguaje *grob* en la plataforma elegida. También se explica cuál sería el mecanismo para portar la implementación a otra plataforma.

4.1. Compilador

El lenguaje utilizado para desarrollar el compilador fue *Haskell*. El compilador se divide en dos fases, la primera implementa un parser que recibe como entrada el programa en lenguaje FROB y genera un árbol de sintaxis abstracta (AST). En Haskell se define el AST como un tipo de datos. Luego se recorre el AST y se genera como salida un archivo con las instrucciones en bajo nivel.

4.2. Máquina virtual

El lenguaje de programación para el desarrollo de la máquina virtual es $C++$ ya que es posible compilarlo para casi cualquier plataforma objetivo. Existen dos limitaciones importantes a tener en cuenta, la primera es que el espacio de memoria varía en diferentes plataformas, por lo que se desea sea posible compilar la máquina aún con un espacio muy reducido. La segunda es que las plataformas varían en capacidades de *Entrada/Salida*, es importante que quien compila la máquina y arma un entorno tenga conocimiento de cómo disponer las mismas y qué limitaciones existen, por ejemplo: Cantidad de pines digitales o analógicos.

La implementación modelo, se hizo utilizando la plataforma *MBED LPC1768*, se puede encontrar documentación de la misma en [4] y en [5].

Capítulo 5

Casos de estudio

En esta sección veremos un caso de estudio usado para verificar la implementación.

5.1. Problema

5.2. Solución

```
distance :: Event Number
color_r :: Event Number
color_l :: Event Number

type Status = Following,

delivery = {
  speed_l = 0,
  speed_r = 0,
  faltan_casas

rises :: Event Number -> Event Number
rises = lift fst $ foldE (\(_, last) new -> if last < new then (1, new) else (0, new

main = let viendo_casa = mapE (hay_casa) distance,
           nueva_casa = rises viendo_casa,
           cuenta = foldE (+) 0 nueva_casa,
           llegue = mapE (>= CASA_ENTREGAR) cuenta,
```

5.3. Conclusiones del caso

Capítulo 6

Conclusiones

Intro .... que se hizo, puntos a favor, etc..

6.1. Trabajo futuro

El principal trabajo futuro sería ...

Bibliografía

[1] John Peterson, Paul Hudak, and Conal Elliott. Lambda in motion: Controlling robots with Haskell. In *Practical Aspects of Declarative Languages*, 1999.

[2] Evan Czaplicki. Elm: Concurrent frp for functional guis. Master's thesis, Harvard University, mar 2012.

[3] Sitio web de Yampa. urlhttps://wiki.haskell.org/Yampa. Último acceso: 29/04/2015.

[4] Sitio web de MBED-LPC1768. `http://developer.mbed.org/platforms/mbed-LPC1768`. Último acceso: 30/04/2015.

[5] Sitio web de MBED. `https://mbed.org`. Último acceso: 30/04/2015.

Apéndices