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

Proyecto de grado, Facultad de Ingeniería, Universidad de la República

Guillermo Pacheco

Tutores: Marcos Viera, Jorge Visca, Andrés Aguirre

10 de diciembre 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 cual 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, no familiarizados con la electrónica ni la informática.

Para resolver el problema, el mismo se dividió en tres etapas. La primera consistió en la definición del lenguaje Willie de alto nivel funcional y reactivo.

Luego se definió el lenguaje Alf de bajo nivel (*Bytecode*) más simple de interpretar y se implementó usando el lenguaje *Haskell* un compilador que traduce un programa Willie al lenguaje Alf.

La última etapa consiste en crear una máquina virtual, que sea capaz de interpretar el lenguaje Alf. Por cada plataforma objetivo, es posible realizar una implementación de la máquina, lo que permite ejecutar un mismo programa en alto nivel en diferentes plataformas.

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 esta forma se creó un lenguaje reactivo con las características deseadas y se codificó una máquina virtual que permite su ejecución en una arquitectura objetivo deseada.

Las implementaciones tanto de la máquina virtual como del compilador son fáciles de mantener, portables y al ser modulares cuentan con la flexibilidad necesaria para garantizar su extensibilidad.

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*. El proyecto es motivado por la necesidad de contar con un lenguaje que brinde soporte al uso de la robótica como herramienta educativa.

La característica más destacable es utilizar un enfoque orientado a programación funcional, y lograr con ello resolver problemas de robótica razonando en alto nivel.

Para analizar el problema, se desarrolla un estado del arte relevando un conjunto de lenguajes de programación funcional reactiva y se introducen los conceptos principales.

Luego se analiza un conjunto de plataformas de hardware con bajas capacidades de cómputo, en lo posible de bajo costo, que puedan ser utilizadas para construir robots autónomos.

Se define el lenguaje Willie funcional reactivo de alto nivel, que permite expresar los comportamientos de un robot y sus interacciones. El lenguaje permite expresar los mismos en base a *Señales* y relaciones entre ellas, que permiten capturar la naturaleza reactiva del dominio. Al ser construído para enseñar conceptos de robótica, uno de los objetivos es que sea simple de utilizar e intuitivo.

Luego se muestra el proceso desde que se escribe un programa Willie hasta que es ejecutado dentro de una plataforma de hardware de las anteriormente relevadas.

Para cumplir con el objetivo de que la implementación sea portable, se separó la misma en dos fases. Los programas Willie son compilados a código Alf de menor nivel de abstracción, el cual 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 construye

un robot físico y se implementa usando Willie la solución a un desafío robótico tomado de la competencia *SumoUY* [4].

Capítulo 2

Programación Funcional Reactiva

2.1 Programación Funcional Reactiva

Tradicionalmente los programas reactivos se escriben como una secuencia de acciones imperativas. Existe un ciclo de control principal, donde en cualquier momento se leen valores de las entradas, se procesan, se mantiene un estado y se escriben valores en las salidas.

Los programas también suelen formarse por eventos y código imperativo que se ejecuta cuando un evento ocurre.

Dicho código imperativo suele hacer referencia y manipular un estado global desde diferentes rutinas.

Esta forma de programación tiene como consecuencia que como un valor puede ser manipulado desde diferentes lugares, puedan producirse problemas de concurrencia o llegar a un estado global inconsistente.

En el paradigma FRP no hay un estado compartido explícito, un programa se forma con valores dependientes del tiempo cuya única forma de ser modificados es a partir de su definición, preservando la consistencia.

Un 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 inicial y producen una salida completa al finalizar su ejecución. Por ejemplo un compilador.

2.1.1 FRP Clásico

El paradigma FRP comenzó a ser utilizado por Paul Hudak y Conal Elliot en Fran (Functional Reactive Animation [5]) para crear animaciones interactivas

de forma declarativa.

Su implementación está embebida en el lenguaje Haskell.

Los programas funcionales puros, no permiten modificar valores, sino que una función siempre retorna el mismo valor dadas las mismas entradas, sin causar efectos secundarios.

Esta propiedad es deseable para fomentar la reutilización del código pero no ayuda a mantener un estado. En la programación reactiva, es necesario mantenerlo por ejemplo para saber la posición del puntero del mouse en una interfaz, o para saber la ubicación de un robot.

En FRP para representar estado, éste se modela como valores dependientes del paso del tiempo.

Para esto, Fran define dos abstracciones principales, que son *Eventos* y *Comportamientos*.

Definición 1. *Comportamiento (Behaviour). [6]* [1]

Un comportamiento es una función que dado un instante de tiempo retorna un valor.

$$type\ \boldsymbol{Behaviour}\ \alpha = \boldsymbol{Time} \rightarrow \alpha$$

Los comportamientos son muy útiles al realizar animaciones, para modelar propiedades físicas como velocidad o posición. Esta abstracción permite que el desarrollador solo se ocupe de definir cómo se calcula un valor, sin implementar la actualización del mismo y dejando esos detalles al compilador.

Ejemplos de comportamientos aplicados a robótica pueden ser:

- *entrada* sensor de distancia, temperatura, video.
- *salida* velocidad, voltaje.
- *estado* explícito como saber que tarea se está haciendo.

Ejemplos de funciones que se pueden aplicar a los comportamientos incluyen:

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

[1] La definición de comportamientos en Fran no coincide con la definición de comportamiento normalmente utilizada en robótica. En bibliografía posterior, comportamientos fue cambiado por señales para evitar ésta ambiguedad.

Por ejemplo dado el comportamiento `time` que representa el tiempo, dada una aceleración constante `a` de un objeto, se puede definir la velocidad en función del tiempo como:

$$\texttt{speed} = \texttt{a} * \texttt{time}.$$

A su vez se puede definir la posición en función del tiempo como:

$$\texttt{position} = \texttt{speed} * \texttt{time}.$$

Definición 2. *Eventos. (Events)[6]*

Los eventos representan una colección discreta finita o infinita de valores junto al instante de tiempo en el que cada uno ocurre.

$$\textit{type } \boldsymbol{Events}\ \alpha = [(\boldsymbol{Time}, \alpha)]$$

Los eventos se utilizan para representar entradas discretas como por ejemplo cuando una tecla es oprimida, o cuando se recibe un mensaje o una interrupción.

También pueden ser generados a partir de valores de un comportamiento, como ser *Temperatura alta*, *Batería baja*, etc.

Se puede generar nuevos eventos a partir de eventos existentes, por ejemplo aplicando funciones a eventos, o seleccionando ciertos valores usando las funciones `map` y `filter`:

- *map* Obtiene un nuevo evento aplicando una función a un evento existente.
- *filter* Selecciona valores que son relevantes.

Frob

Utilizando como base el trabajo realizado en Fran, se construyó Frob (Functional Robotics [6] [7]) un lenguaje funcional reactivo embebido en el lenguaje `Haskell`, aplicado al dominio de la robótica.

En éste trabajo se introdujo el concepto de reactividad con el cuál utilizando los conceptos de comportamientos y eventos, éstos se combinan para realizar las tareas que un robot debe hacer. La estrategia que presenta, es de formalizar las tareas por medio de comportamientos, y conseguir que los comportamientos se modifiquen utilizando eventos y un conjunto de combinadores específicos.

Un ejemplo es, dado un robot, este se tiene que mover a una velocidad constante hasta que se supere un tiempo máximo o se detecte un objeto. En Frob se expresaría de esta manera:

```
goAhead :: Robot -> Time -> WheelControl
goAhead r t =
  (forward 30 'untilB'
    (predicate (time > t) .|.  predicate (frontSonar r < 20))
     -=> stop)
```

Lo que se leería cómo: "Para el robot r, moverse hacia adelante a velocidad 30, hasta que se exceda el tiempo t, o se detecte un objeto a menos de distancia 20. En ese momento detenerse."

- `predicate` se utiliza para generar eventos a partir de comportamientos en base a una condición.
- `untilB` cambia de comportamiento en respuesta a un evento.
- `.|.` toma dos eventos y los intercala.
- `-=>` Asocia un nuevo valor, luego de que ocurre el evento.

Otro punto importante de Frob, es que los periféricos del robot se asumen implementados, permitiendo que el desarrollador se concentre en la lógica específica de su problema, y no en resolver problemas del hardware. Además la lógica de leer las entradas, procesarlas y escribir las salidas es realizada por el flujo de control de Frob, y no de cada programa. En la implementación utilizaron un esquema simple, donde se leen todos los valores de todas las entradas y se procesan lo más rápido posible. Está claro que no es la mejor estrategia, porque puede causar demoras en el procesamiento y los datos leídos son válidos por un período corto de tiempo.

2.1.2 Real-Time FRP y Event-Driven FRP

Si bien el paradigma clásico de FRP permite expresar naturalmente programas reactivos, ésta expresividad no es gratuita, sino que puede llevar a errores muy difíciles de encontrar en los programas. Un ejemplo de ésto es el llamado *time leak*, al implementarse en un lenguaje como Haskell, que cuenta con evaluación a demanda, los cálculos a demanda sobre los *Behaviours* puede que se retrasen y se acumulen, y al momento de necesitarse, el cálculo es tan largo que deja al programa sin memoria o afecta demasiado el tiempo de respuesta.

También pueden ocurrir *space leaks* donde un cálculo se retrasa indefinidamente, y la acumulación de los mismos consume el total de la memoria.

Como solución a ésto se propuso Real-Time FRP [8], una simplificación que garantiza mayor eficiencia. Utilizando el tipo de datos paramétrico Maybe [2], se realiza un isomorfismo entre Events y Behaviour.

Definición 3. *Isomorfismo en RT-FRP entre Events y Behaviour.*

$$\textbf{\textit{Events}}\ \alpha \approx \textbf{\textit{Behaviour}}\ (\textbf{\textit{Maybe}}\ \alpha)$$

Utilizando ésta simplificación, se agrupan las dos definiciones en un nuevo tipo llamado `Signal`.

Definición 4. *Señal (Signal).*

$$\textit{type}\ \textbf{\textit{Signal}}\ \alpha = \textbf{\textit{Time}} \rightarrow \alpha$$

Para garantizar las restricciones de que el tiempo y el espacio requerido por los programas es acotado, RT-FRP define un lenguaje base (lambda cálculo) de alto órden, y luego sobre esa base define un lenguaje reactivo que obliga a declarar las señales y sus conexiones.

Sobre este lenguaje restringido, se proveen demostraciones de que se cumplen las restricciones.

Event-Driven FRP Poco después de ésto, se propuso *Event-Driven FRP* [9], otra simplificación que añade como restricción que las señales sólo puedan ser modificadas mediante eventos discretos, en lugar de ser contínuamente actualizadas con el paso del tiempo.

Aunque parezca muy restrictiva, la justificación de la misma es que los sistemas reactivos que se desean implementar están fuertemente guiados por eventos.

La propuesta de Event-Driven FRP es llevar la programación reactiva a microcontroladores, para lo cuál define un lenguaje imperativo llamado *Simple C*, de tal forma que a partir del mismo sea muy simple compilarlo al lenguaje *C* o a las variantes que existen para microcontroladores.

En particular se implementó un prototipo que es capaz de generar código para el microcontrolador *PIC16C66* [10].

2.1.3 Arrows

Arrowized FRP (AFRP [11] [12]) intenta resolver los problemas de la FRP Clásica cambiando la forma en la que se crean los programas. En éste paradigma, tampoco se toman en cuenta por separado los eventos, se utiliza la misma definición de señal que en RT-FRP.

[2] El tipo `Maybe` α en Haskell tiene dos valores posibles, `Just` α y `Nothing`.

Definición 5. *Señal (Signal).*

$$\textbf{\textit{Signal}}\ \alpha = \textbf{\textit{Time}} \rightarrow \alpha$$

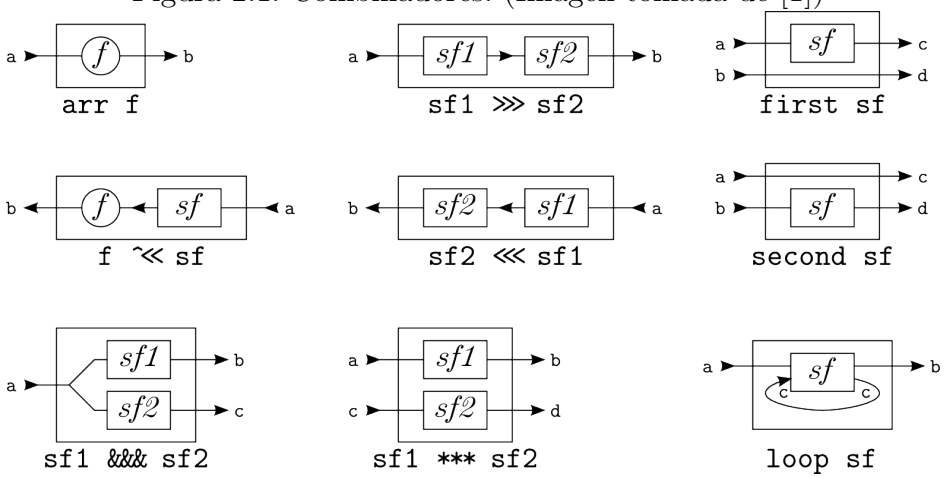
En lugar de permitir que se manipulen las señales como valores de primera clase, ésto se prohibe y se define el concepto de *Signal Functions* (funciones sobre señales). El programador tendrá acceso sólo a las señales utilizándolas.

Definición 6. *Funciones sobre señales (Signal Functions).*

$$\textbf{\textit{SF}}\ \alpha\ \beta = \textbf{\textit{Signal}}\ \alpha \rightarrow \textbf{\textit{Signal}}\ \beta$$

Sin embargo, la representación de `SF` no es accesible al desarrollador, en lugar de eso se define un conjunto de funciones sobre señales primitivas y un conjunto de combinadores para componerlas.

Figura 2.1: Combinadores. (Imagen tomada de [1])



AFRP está basado en *Arrows* [13] una generalización del concepto de *Monads*. En particular `SF` es una instancia de la clase *Arrow*.

Un programa es una `SF` global, compuesta por un conjunto de otras *Signal functions*, y un intérprete corre ésta instancia global.

En la Figura 2.1 se puede ver un conjunto extenso de combinadores, sin embargo se puede definir un conjunto minimal utilizando sólo los combinadores `arr`, $\gg$ y `first`, aunque no es el único.

Yampa El framework *Yampa* [1] embebido en Haskell, define los operadores de la Figura 2.1 para combinar señales y aplicar funciones sobre ellas.

2.1.4 Elm: Programación funcional reactiva concurrente

Por último existe un lenguaje llamado Elm [2], creado para poder escribir aplicaciones web reactivas de una forma funcional declarativa. En Elm, las entradas se asumen conocidas y son dadas, por ejemplo un click o el movimiento del mouse, o una tecla presionada.

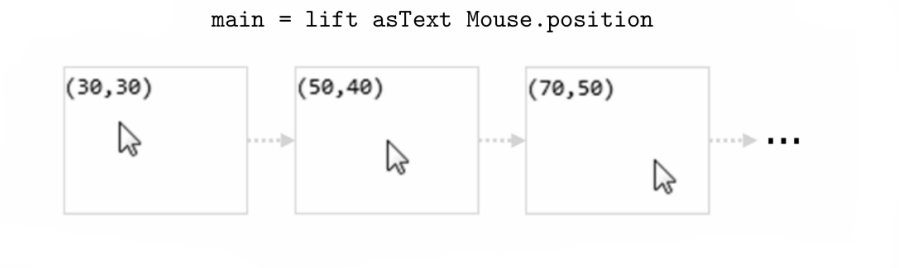
Todos los valores son señales, el combinador $lift$ toma una señal y una función, y define otra señal resultado de la aplicación de la función sobre cada valor de la señal.

Para combinar más de una señal, se usa el combinador $lift_2$ o $lift_n$ con $n \in N$. En los casos que se desea tener memoria, por ejemplo llevar una cuenta de ocurrencias de una señal, o mantener un estado explícito, se utiliza el combinador $foldp$.

El combinador $foldp$ opera sobre una señal como el operador $fold$ sobre una lista. Dado un valor inicial y una función, aplica la función sobre cada valor, utilizando el último valor conocido como primer argumento.

La Figura 2.2 muestra un programa Elm de ejemplo. El mismo toma una señal `Mouse.position`, y dada la función `asText` que convierte un par de valores en texto, utiliza `lift` para aplicar la función a las coordenadas de un Mouse.

Figura 2.2: Ejemplo de programa Elm. (Imagen tomada de [2])



2.2 Simplificación del paradigma

Al intentar implementar en un computador programas utilizando éste paradigma, nos encontramos con varias limitaciones. Una de ellas es que no es posible tener valores que se modifiquen de forma continua, la capacidad de cómputo es finita, y no es posible ejecutar tareas al mismo tiempo, inclu-

so en un entorno con paralelismo, el mismo está acotado a la cantidad de procesadores con los que se cuente.

Aunque el paradigma distinga valores continuos de valores discretos, se puede hacer una simplificación y asumir que todos los valores son señales. Las señales en realidad serán una secuencia de valores en el tiempo, las señales que dependan de otras serán notificadas en el momento que éstas cambien.

Valores como el tiempo (reloj) y otros sensores como ser un sensor de distancia, entregarán valores periódicamente. Sería imposible que un programa reaccione a un valor continuo, sin embargo, es muy fácil asumir que lo que importa del tiempo en realidad es dada una variación de tiempo cómo se debe comportar nuestro programa.

A su vez un sensor de distancia no nos puede entregar infinitos valores, y generalmente sólo importa poder recibir un valor nuevo en un período relativamente corto de tiempo. Al igual que una película que simula ser continua, en realidad está compuesta de una serie discreta de imágenes en el tiempo, un robot reacciona de manera discreta a los cambios que detecte en sus actuadores, simulando reacción en tiempo continuo.

Otra limitación en nuestro caso, al tratarse de robots con bajas capacidades de cómputo, es que la implementación del lenguaje no siempre tendrá más de un procesador disponible. Para simular paralelismo se asumirá que el tiempo que se necesita para hacer cálculos es de una magnitud mucho menor al tiempo que transcurre para recibir un nuevo dato de una entrada, o enviar un dato a una salida.

Se implementará una máquina capaz de correr dichos programas, la cuál esperará por valores en todas las entradas, y en base a los mismos, actualice todos los eventos que dependan de ellas, y luego actualice las salidas que dependan de éstos.

Cada actualización se hará secuencialmente hasta terminar, y en caso de contar con más de un procesador, se asignarán en paralelo las actualizaciones utilizando varios hilos de ejecución.

2.3 Ventajas

La motivación para utilizar el paradigma presentado, es que un programa reactivo escrito de forma iterativa es susceptible a contener errores de concurrencia al modificar valores en diferentes rutinas. A su vez, es difícil estructurar un programa iterativo para que reaccione rápidamente a los cambios.

Un patrón utilizado comunmente para estructurar un programa reactivo es el patrón `Observer`. En dicho patrón, un `Sujeto` puede ser observado por un `Observador`, éste último se subscribe al `Sujeto`, el cuál notifica a todos

sus `Observadores` cuando su valor cambia.

La desventaja de hacer un programa reactivo siguiendo ese esquema, es que es difícil ver en que momento ocurren las actualizaciones de los `Observadores`. Si hay muchos `Observadores` suscritos a cambios de muchos `Sujetos`, se vuelve complejo mantener el código al tener tantas interacciones implícitas.

Al usar un lenguaje funcional, las interacciones son especificadas declarativamente, y se puede entender como un valor es formado a partir de otros.

Además se cumple que una función invocada con las mismas entradas en diferentes instantes de tiempo, siempre retorna el mismo valor, lo que ayuda a razonar sobre un programa.

Para ver que es lo que está sucediendo en un programa se pueden obtener los valores de todas las entradas en un instante de tiempo, como si fuera una fotografía y evaluar las señales para encontrar errores o corroborar si el mismo es correcto.

De ésta manera si un programa tiene un error, se puede tomar una secuencia de instantes, como si fuera una grabación y entender donde está el problema claramente, sin necesidad de seguir varios hilos de ejecución ni razonar sobre la concurrencia.

Capítulo 3

Plataformas de Hardware

En esta sección se describen las plataformas de hardware relevadas durante el estado del arte, junto con sus características. Las características principales estudiadas son el espacio de almacenamiento, el lenguaje de programación o herramientas estándar de cada plataforma, si es un proyecto libre, el procesador y la arquitectura.

A continuación se presentan las diferentes familias de plataformas de hardware estudiadas. El objetivo es obtener una medida de las capacidades de cómputo de las plataformas y evaluar las herramientas de desarrollo que se pueden utilizar. Las características estudiadas permiten elegir un lenguaje para desarrollar común entre las plataformas, y permiten tener una medida como límite del tamaño de los programas que se podrán implementar.

3.1 Arduino

Arduino [14] es una plataforma abierta de prototipado, basada en software y hardware flexible fácil de usar. Está pensada para ser usada por diseñadores, artistas, como hobby, para crear objetos y ambientes interactivos. Entre sus productos, hay placas y kits de componentes. Los kits de arduino generalmente tienen interfaz usb con soporte para programarlo usando la propia placa sin necesidad de un programador por hardware.

También los pines de entrada/salida del microprocesador están diseñados para poder colocar fácilmente cables y conectar periféricos sin necesidad de soldar.

Tambien incluyen leds y botones para resetear la placa o utilizarlos como sensor.

Existe un entorno de desarrollo integrado (IDE) que utiliza una implementación del compilador `gcc` [15] para la arquitectura `avr` [16] de *Atmel*

[17] y puede ser utilizado para programar sobre los kits.

Variedad de bibliotecas y abstracciones de sensores, actuadores y protocolos de comunicación, ya están implementados y pueden ser usados en los kits. Al ser un proyecto libre las bibliotecas son publicadas y mantenidas por una comunidad abierta.

La arquitectura usada por casi todos los kits es `avr Atmel` pero existen algunos con arquitectura `ARM` [18].

El lenguaje estándar para desarrollar programas se llama `Arduino`, sin embargo el lenguaje es `C/C++`, cambiando la forma en que se invoca el programa principal y con algunas funciones y formato predefinido.

La Tabla 3.1 muestra un listado de los modelos de Arduino, cuánta memoria persistente tienen (Flash) en kilobytes, con cuánta memoria RAM cuentan, cuánta memoria EEPROM tienen en kilobytes, que procesador tienen y la frecuencia de funcionamiento.

Salvo el modelo `Due`, el resto utilizan la arquitectura `avr` de 8 bit. La memoria ram varía entre 16 y 512 kilobytes. Los modelos más populares y representativos, son el Arduino `Uno` y el Arduino `Nano 328`, ambos con 32 KB de memoria Flash, 2 KB de memoria RAM (SRAM) y procesador `ATmega328` a 16 MHz.

3.2 Mbed

Mbed, al igual que Arduino es una plataforma abierta de prototipado, su objetivo es que se puedan desarrollar prototipos en un tiempo corto.

Cuenta con herramientas colaborativas como ser un entorno de desarrollo integrado (IDE) web, interfaz web de control de versiones, donde se pueden publicar proyectos, extender y colaborar con proyectos de otros usuarios.

Existe una gran variedad de bibliotecas desarrolladas para los kits Mbed, que al igual que en Arduino, implementan funcionalidades básicas como ser protocolos de comunicación e interacción con componentes externos.

La arquitectura usada por Mbed es `ARM`, principalmente `ARM Cortex-M3` y `ARM Cortex-M0`.

El compilador web es práctico para colaborar con otros usuarios y no tener que armar un entorno local. Las aplicaciones pueden ser cargadas en las placas usando el entorno web sin necesitar instalación del compilador.

El lenguaje utilizado es `C/C++` con bibliotecas especializadas de Mbed. Mbed también cuenta con un HDK (Hardware development kit) para diseño de hardware especializado, luego de prototipar.

En la Tabla 3.2 se muestra un listado con los modelos más relevantes y sus características. La memoria RAM varía entre 16 kilobytes y 1 megabyte.

Cuadro 3.1: Modelos arduino

Modelo	Flash	SRAM (kb)	EEPROM (kb)	Procesador	Arquitectura	Frecuencia
Uno	32 KB	2	1	ATmega328	8 bit AVR	16 MHz
Leonardo	32 KB	2.5	1	ATmega32u4	8 bit AVR	16 MHz
Due	512 KB	96	-	AT91SAM3X8E	ARM Cortex-M3	84 Mhz
Yun	32 KB	2.5	1	ATmega32u4	8 bit AVR	16 MHz
Tre	32 KB	2.5	1	ATmega32u4	8 bit AVR	16 MHz
Micro	32 KB	2.5	-	ATmega32u4	8 bit AVR	16 MHz
Robot	32 KB	2.5	1	ATmega32u4	8 bit AVR	16 MHz
Esplora	32 KB	2.5	1	ATmega32u4	8 bit AVR	16 MHz
Mega ADK	256 KB	8	4	ATmega2560	8 bit AVR	16 MHz
Ethernet	32 KB	2	1	ATmega328	8 bit AVR	16 MHz
Mega 2560	256 KB	8	4	ATmega2560	8 bit AVR	16 MHz
Mini	32 KB	2	1	ATmega328	8 bit AVR	16 MHz
LilyPad USB	32 KB	2.5	1	ATmega32u4	8 bit AVR	8 Mhz
LilyPad Simple	32 KB	2	1	ATmega328	8 bit AVR	8 Mhz
LilyPad (168V)	16 KB	1	512 Bytes	ATmega168V	8 bit AVR	8 Mhz
LilyPad (328V)	16 KB	1	512 Bytes	ATmega328V	8 bit AVR	8 Mhz
Nano (168)	16 KB	1	512 Bytes	ATmega168	8 bit AVR	16 MHz
Nano (328)	32 KB	2	1	ATmega328	8 bit AVR	16 MHz
Pro mini (3.3v)	16 KB	1	512 Bytes	ATmega168	8 bit AVR	8 Mhz
Pro mini (5v)	16 KB	1	512 Bytes	ATmega168	8 bit AVR	16 MHz
Pro (168)	16 KB	1	512 Bytes	ATmega168	8 bit AVR	8 Mhz
Pro (328)	32 KB	2	1	ATmega328	8 bit AVR	16 MHz
Fio	32 KB	2	1	ATmega328P	8 bit AVR	8 Mhz

Sin embargo, el modelo más popular es el `LPC1768` con 512 KB de Flash y 64 KB de memoria RAM.

En comparación con `Arduino`, generalmente se cuenta con mayor cantidad de memoria y capacidad de procesamiento.

Este modelo será utilizado para la implementación, al no ser tan reducido, se puede crear una implementación modelo para el mismo, y luego evaluar si es posible reducir el tamaño para trabajar con modelos con menor capacidad de cómputo.

Cuadro 3.2: Modelos Mbed

Modelo	Flash	RAM (KB)	Procesador	Frecuencia
NXP LPC1768	512 KB	64 (sram)	ARM Cortex-M3	96 MHz
NXP LPC11U24	32 KB	8	ARM Cortex-M0	48 MHz
Freescale FRDM-KL25Z	128	16 KB	ARM Cortex-M0+	48 MHz
NXP LPC800-MAX	16 KB	4	ARM Cortex-M0+	30 MHz
NXP EA LPC4088	512 KB	96 (sram)	ARM Cortex-M4	120 MHz
NXP DipCortex M0	32 KB	8	ARM Cortex-M0	50 MHz
NXP DipCortex M3	64 KB	12	ARM Cortex-M3	72 MHz
NXP BlueBoard-LPC11U24	32 KB	8	ARM Cortex-M0	48 MHz
NXP WiFi DipCortex	64 KB	12	ARM Cortex-M3	72 MHz
NXP Seeeduino-Arch	32 KB	8	ARM Cortex-M0	48 MHz
NXP mbed LPC1114FN28	32 KB	4	ARM Cortex-M0	50 MHz
Ublox U-blox C027	512 KB	32	ARM Cortex-M3	96 MHz
NXP EA LPC11U35	64 KB	10	ARM Cortex-M0	48 MHz
ST Nucleo F103RB	128 KB	20 (sram)	ARM Cortex-M3	72 MHz
Freescale FRDM-KL46Z	256 KB	32	ARM Cortex-M0+	48 MHz
NXP Seeeduino-Arch-Pro	512 KB	32	ARM Cortex-M3	96 MHz
ST Nucleo F302R8	64 KB	16 (sram)	ARM Cortex-M4	72 MHz
ST Nucleo L152RE	512 KB	80 (sram)	ARM Cortex-M3	32 MHz
ST Nucleo F401RE	512 KB	96 (sram)	ARM Cortex-M4	84 MHz
ST Nucleo F030R8	64 KB	8 (sram)	ARM Cortex-M0	48 MHz
Freescale FRDM-K64F	1 MB	256	ARM Cortex-M4	120 MHz
Nordic nRF51822	128 KB	16	ARM Cortex-M0	16 MHz
FRDM-KL05Z	32 KB	4	ARM Cortex-M0+	48 MHz
LPCXpresso1549	256 KB	36	ARM Cortex-M3	72 MHz
LPCXpresso11U68	256 KB	36	ARM Cortex-M0+	50 MHz

3.3 Robotis

La empresa `Robotis` desarrolla robots para uso educativo, así como una gama de robots para uso competitivo. Los kits de Robotis están diseñados para uso final, es decir, proveen los controladores, así como los componentes para armar la estructura, sensores y actuadores.

Para el uso de los kits se deben utilizar las herramientas de desarrollo de Robotis.

No es un proyecto abierto, por lo que no puede ser fácilmente extendido, ni modificado. Es posible programar utilizando `C` embebido, descargando los archivos fuente para sus plataformas, aunque es un proceso bastante complejo y no cuenta con buena documentación.

Robotis cuenta con un lenguaje llamado `Task` y se necesita un entorno de desarrollo integrado propietario llamado `IDE Roboplus` para utilizarlo. En el `IDE` se pueden generar tareas y programar movimientos del robot en base a movimiento de motores y un diseño tridimensional gráfico.

Los controladores utilizados se llaman `CM-510`, `CM-530` y `CM-100A`. Algunos con arquitectura AVR y otros con ARM internamente.

El `CM-510` contiene un microcontrolador `ATMega128`, el `CM-530` un `ARM Cortex-M3` y el `CM-100A` un microcontrolador ATMega8.

En la Tabla 3.3 se pueden ver las especificaciones técnicas de los diferentes kits de *Robotis*.

Cuadro 3.3: Modelos Robotis

Modelo	Flash	RAM (kb)	EEPROM (kb)	Procesador	Arquitectura	Frecuencia
CM-100A	8 KB	1 (sram)	512	ATmega8	8 bit AVR	16 MHz
CM-5	128 KB	4 (sram)	4	ATmega128	8 bit AVR	16 MHz
CM-510	256 KB	8 (sram)	8	ATmega2561	8 bit AVR	16 MHz
CM-530	512 KB	64	-	STM32F103RE	ARM Cortex-M3	72 MHz
CM-700	256 KB	8 (sram)	8	ATMega2561	8 bit AVR	16 MHz
CM-730	512 KB	64	-	STM32F103RE	ARM Cortex-M3	72 MHz
CM-900	64 KB	20 (sram)	-	STM32F103C8	ARM Cortex-M3	72 MHz

3.3.1 Bioloid STEM

Creado para uso educativo y competencias robóticas. El kit provee el hardware y clases enseñando a construir distintos robots para distintos usos, involucrando conceptos de ciencias, tecnología, ingeniería y matemáticas.

Utiliza el controlador `CM-530` internamente.

Cuenta con un conjunto de componentes que son:

- Sensor Infrarrojo
- Array de 7 sensores infrarrojos (detectan objetos)
- Control remoto y receptor
- 6 motores dinamixel
- Piezas para crear estructura de un robot

3.3.2 Bioloid Premium

Diseñado para educación, competiciones y entretenimiento. Se pueden construir variedad de robots como humanoide y animales. El kit contiene 29 ejemplos de robot y programas de ejemplo. Utiliza el controlador `CM-530`.

Incluye los siguientes componentes:

- 18 motores dinamixel (AX-12A)
- Sensor giroscópico
- Receptor infrarrojo
- Control remoto y receptor
- Sensor de distancia
- Sensor infrarrojo para detección de objetos.
- Piezas para crear estructura de robot.

3.3.3 Bioloid GP

Humanoide optimizado para competencias robóticas. Esqueleto liviano y resistente. Instrucciones para jugar al fútbol y hacer tareas de recolección preprogramadas. Ajuste automático de postura con sensor giroscópico.

Utiliza el controlador `CM-530`.

Cuenta con los siguientes componentes:

- 18 motores dinamixel
- Sensor giroscópico
- Control remoto y receptor
- Sensor de distancia
- Piezas de aluminio.

3.4 Butiá

USB4Butiá es una plataforma que surgió de un proyecto de grado de Facultad de Ingeniería – UdelaR. Como característica principal es un kit económico y con un diseño abierto.

El diseño de la placa está publicado junto con instrucciones para construirla, además sus componentes se seleccionaron por ser económicos y de fácil acceso en el medio local.

La placa está pensada principalmente para aumentar capacidades sensoriales y de actuación del robot Butiá [19].

Utiliza el microprocesador `PIC 18F4550` [20] el cuál se programa utilizando el lenguaje `C`.

Además de la placa `USB4Butia`, también existen tres versiones del robot Butiá, así como varias herramientas de desarrollo. Entre ellas se encuentran el entorno de desarrollo integrado `Tortubots` [21], `Butialo` [22] que permite programar utilizando el lenguaje *Lua*, se puede utilizar *Python*, y también otro IDE llamado `Yatay` [23] también desarrollado como proyecto de la Facultad de Ingeniería.

La versión actual de la placa es `USB4Butia 1.9`. Todas las versiones del proyecto cuentan con todo el software necesario libre, así como las instrucciones para construir el hardware, que también es libre.

El robot `Butia 1.0` utiliza la placa `USB4Butia 1.9`, la versión `Butia 2.0` utiliza una placa *Arduino*, y la versión `Butia 3.0` utiliza la placa `Beagleboard Black` [24].

En la Tabla 3.4 se muestran las especificaciones del `UBS4Butia 1.9`, ya que *Arduino* ya fue relevado y `Beagleboard Black` no es una plataforma con bajas capacidades de cómputo.

Cuadro 3.4: Modelo USB4Butia

Modelo	Flash	SRAM	EEPROM	Procesador	Arquitectura	Frecuencia
USB4Butia 1.9	32 KB	2 KB	256 Bytes	PIC 18F4550	8 bit	48 MHz

3.5 Conclusiones

Luego de estudiar las plataformas, se decide utilizar el lenguaje `C` con extensiones de `C++` para facilitar la portabilidad entre plataformas. Al planificar la implementación, se toma en cuenta el objetivo de tener un tamaño de programa menor a 64 KB y en lo posible menor a 32 KB. Se deberá intentar utilizar memoria estática en lo posible, en lugar de memoria dinámica, ya

que no todas las plataformas cuentan con las bibliotecas estándar necesarias para su manejo.

Los proyectos libres mejoran la portabilidad, ya que cuentan con más y mejor documentación, y una comunidad abierta. En el caso de particular de `Robotis` se evaluó la posibilidad de desarrollar utilizando `C++` y si bien es posible, no cuenta con documentación más que el propio código fuente.

En el resto de los proyectos, se cuenta con documentación y una comunidad abierta que apoya la realización de proyectos que extiendan sus funcionalidades.

Capítulo 4

Diseño

Dados los requerimientos específicos del dominio, se decidió definir un lenguaje propio llamado Willie. El lenguaje comparte con el Real-Time FRP y con el Arrowized FRP la noción de señal, donde no se distingue entre eventos discretos y continuos. A su vez, en el lenguaje se permite manipular las señales directamente, con un conjunto de primitivas similar al del lenguaje Elm.

En este capítulo se describe el diseño del lenguaje Willie junto con su semántica. Luego se explica de qué manera es traducido al lenguaje Alf de bajo nivel, mas simple de interpretar, el cual podrá ser interpretado por implementaciones de una misma máquina virtual en diferentes plataformas de hardware.

También se describirán las etapas de compilación, desde que se escribe un programa en alto nivel hasta que es ejecutado en una plataforma objetivo.

El diagrama de la Figura 4.1 resume todas las etapas y componentes necesarios.

En la parte de arriba de la Figura se ve un programa en el lenguaje Willie de alto nivel. El desarrollador escribe dicho programa y ejecuta el compilador `williec` y obtiene un archivo Alf binario.

Debajo se muestran diversas plataformas, cada una con su implementación de la máquina virtual Alfvm instalada.

El desarrollador podrá cargar el mismo código Alf en cualquier robot que esté construido utilizando cualquiera de las plataformas, y la máquina virtual se encargará de interpretarlo.

Figura 4.1: Etapas y componentes

4.1 Lenguaje Willie

En Willie los programas consisten de un conjunto de funciones, y un conjunto de aplicaciones de primitivas del paradigma de programación funcional reactiva.

Es un lenguaje funcional, tipado y con inferencia de tipos. En él todos los valores son inmutables, una vez que son declarados no se pueden modificar. Las funciones declaradas son puras, por lo tanto no pueden tener efectos secundarios.

Para controlar un robot se debe declarar un conjunto de señales utilizando las primitivas de FRP.

Para simplificar la implementación dentro del alcance del proyecto, en el lenguaje solo se permitirán valores naturales ($\mathcal{N}$), y funciones de naturales en naturales. ($\mathcal{N} \to \mathcal{N}$).

El lenguaje evalúa las expresiones, tan pronto como es posible. Las funciones tienen un conjunto de variables libres, cuando un valor les es asignado, el resultado es calculado. Para invocar una función es necesario saturarla, es decir, invocarla con todos los parámetros que declara.

Figura 4.2: Gramatica de Willie

```
program := definitions "do {" frps "}";

definitions := definition | definition definitions;

definition := ident arg_list "=" expr;

frps = frp | frp "," frps;

frp := ident "<- read" expr
     | ident "<- lift" ident ident
     | ident "<- lift2" ident ident ident
     | ident "<- folds" ident value ident
     | "output" expr ident

expr := name
      | number
      | expr binop expr
      | "if" expr "then" expr "else" expr;

arg_list := "" | ident arg_list;

ident := [a-z_A-Z]+;

number := [-+]?[0-9]+;

binop := '+' | '-' | '/' | '*' | 'or' | 'and'
       | '==' | '<=' | '>' | '<' | '<>' | '>=';
```

Un comentario es una línea que comienza con el símbolo '#'. La gramática completa del lenguaje se puede ver en la Figura 4.2.

El sistema de tipos respeta las reglas de inferencia definidas en la Figura 4.3.

4.1.1 Funciones

Las funciones se definen con un nombre, una lista de argumentos y una expresión. Las variables libres de la expresión son sustituídas al evaluar la función.

```
nombre argumento_1 .. argumento_n = expresion
```

Una expresión puede ser un valor primitivo, una expresión aritmética (por ejemplo una suma o multiplicación), la aplicación de una función, o una expresión condicional. Todas las expresiones retornan un valor al evaluarse.

La sintaxis es muy similar a la del lenguaje `Haskell`, aunque no se permiten funciones anónimas.

Para declarar un valor constante simplemente se escribe una función sin argumentos. Por convención se escriben con mayúsculas, pero no es una restricción.

```
NOMBRE_CONSTANTE = valor
```

Una expresión condicional, debe retornar un valor para cada posible resultado de la condición.

En la figura 4.4 se ve la implementación de la función que retorna un número en la sucesión de Fibonacci, utilizando una expresión condicional.

Se puede ver que una función puede invocarse a si misma, está permitida la recursión.

4.1.2 Combinadores de FRP

Un robot cuenta con un conjunto de sensores y un conjunto de actuadores, cada uno identificado con un número entero.

Para especificar el comportamiento de un robot en un programa, se crean señales a partir de las entradas (sensores), se les aplican funciones y combinan utilizando los combinadores de FRP, y se mapean señales a las salidas (actuadores).

Los combinadores `lift`, `lift2` y `folds`, y las primitivas de entrada/salida `read`, `output` se encuentran dentro del bloque `do` del programa.

Con las primitivas de entrada/salida se define cómo se conectan las señales con sensores y actuadores, y con los combinadores se define un grafo de señales que especifica el comportamiento del robot.

Para que un programa sea válido, el grafo de señales debe ser acíclico.

El bloque `do` permite de manera declarativa expresar las relaciones entre las señales y que funciones se deben aplicar.

La máquina virtual que interpreta el programa, será la encargada de darle valores a las señales y actualizarlas, así como actualizar las salidas de acuerdo a que señal está conectada a ellas.

A continuación se presenta el conjunto de primitivas y combinadores.

Read

Para crear una señal a partir de una entrada, se utiliza la primitiva `read`, que asocia la señal a un identificador.

Asumiendo que un robot tiene un sensor de distancia en la entrada `INPUT_DISTANCE` se puede definir una señal `distance`, que contendrá la distancia en centímetros para cada instante de tiempo.

```
distance <- read INPUT_DISTANCE
```

El tipo de la primitiva `read` es:

$$read :: Int \rightarrow Signal\ a$$

Lift

Usando la primitiva `lift` se puede aplicar una función a la señal, y obtener una nueva señal más compleja resultado de la aplicación.

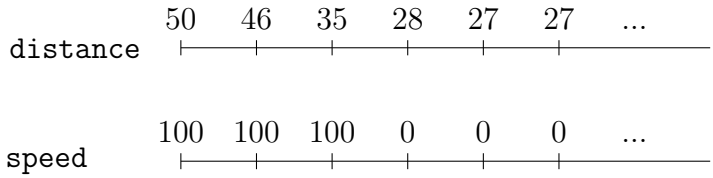
Se puede definir una función `distanceToSpeed` que de acuerdo a una distancia, calcula la velocidad apropiada a la que se debe mover un robot, para detenerse si hay un objeto muy cercano y evitar una colisión.

```
distanceToSpeed n = if (n < MIN_DIST) then STOP else MAX_SPEED
```

Se puede definir la señal `speed`, resultado de aplicar la función `distanceToSpeed` a la señal `distance`.

```
speed <- lift distanceToSpeed distance
```

Se puede ver en una línea de tiempo, los valores que toma cada señal.



El tipo de la primitiva lift es

$$lift :: (a \rightarrow b) \rightarrow Signal\ a \rightarrow Signal\ b$$

Output

La primitiva `output` envía a un actuador el valor de una señal. Asumiendo que el motor del robot está identificado con el valor entero `OUTPUT_ENGINE`:

```
output OUTPUT_ENGINE speed
```

El tipo de la primitiva output es

$$output :: Signal\ a \rightarrow Int \rightarrow ()$$

En la Figura 4.5 se puede ver el ejemplo completo.

LiftN

Para combinar más de una señal, se utiliza la función `lift2` que recibe dos señales y produce una nueva aplicando una función.

$$lift2 :: (a \rightarrow b \rightarrow c) \rightarrow Signal\ a \rightarrow Signal\ b \rightarrow Signal\ c$$

Folds

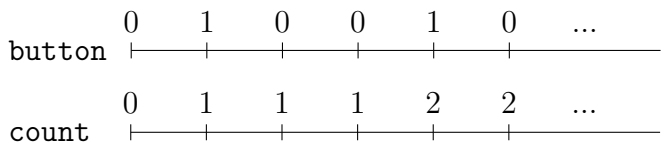
En Willie para mantener un valor que dependa de la historia, se utiliza el combinador `folds`.

El mismo es análogo a la operación `fold` sobre listas, viendo una señal como una lista de valores en el tiempo, el combinador aplica una función al valor actual de una señal y a un valor acumulado. De ésta manera se puede crear una nueva señal para representar estado.

Por ejemplo si se asume definida una señal `button` que tiene el valor 1 cuando se apreta un botón y sino el valor 0, se puede contar cuántas veces se apretó el botón utilizando el combinador `folds` y una función para sumar el valor acumulado y el nuevo.

```
count <- folds suma 0 button
```

Se pueden ver las señales `button` y `count` como los valores en una línea de tiempo:



El tipo del combinador `folds` es:

$$folds :: (a \rightarrow b \rightarrow b) \rightarrow b \rightarrow Signal\ a \rightarrow Signal\ b$$

4.2 Lenguaje de bajo nivel

Al compilar un programa Willie, se obtiene como salida un código intermedio en lenguaje Alf. El mismo es independiente de la plataforma en la que va a ser ejecutado. Para lograr ésto, se define el lenguaje como un conjunto de instrucciones con su semántica y una máquina virtual abstracta que las ejecuta.

Máquina virtual

La máquina que interpreta el lenguaje Alf es una *máquina de stack*.[1].

En una máquina de stack las instrucciones están en notación postfija.[2] Para evaluar expresiones se colocan sus argumentos en una pila, y luego se ejecuta la operación asociada.

Por ejemplo la expresión "$5 + 19 * 8$" en RPN se escribe "5 19 8 * +".

El conjunto *Inputs* representa las entradas de la máquina. Dadas m entradas fijas, cada una se identifica con un entero único entre 1 y $m = |Inputs|$.

Cada $\mathtt{I}_i, i \in (1 \cdots m)$ se corresponderá con un sensor definido en el robot.

Definición 7. *Entradas de la máquina*

$$Inputs \equiv \{\mathtt{I}_1 \cdots \mathtt{I}_m\}.$$

Graficamente se representan con la notación:

También se cuenta con un conjunto *Outputs* de salidas, identificadas de 1 a $k = |Outputs|$.

Cada $\mathtt{O}_i, i \in (1 \cdots k)$ se corresponderá con un actuador del robot.

Definición 8. *Salidas de la máquina*

$$Outputs \equiv \{\mathtt{O}_1 \cdots \mathtt{O}_k\}.$$

Graficamente se representan con la notación:

[1] Stack machine en inglés

[2] RPN (*Reverse polish notation*) del inglés

Las señales que se definan se denotarán $\mathtt{S}_i$, siendo i un índice único que las identifica. El conjunto de las señales se llama *Signals*.

Definición 9. *Señales*

$$Signals \equiv \{\mathtt{S}_1 \cdots \mathtt{S}_s\}.$$

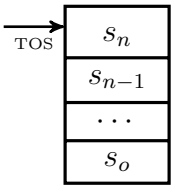
Graficamente se representan con la notación:

La máquina tendrá una pila global, denotada *Stack*. El mismo se representa con una secuencia de valores.

Definición 10. *Pila global*

$$Stack \equiv s_1, \cdots, s_n.$$

El *Stack* se representa graficamente con la notación:



Donde `TOS`[3] indica el índice del tope del mismo. Se cumple que $Stack_{TOS} = s_n$.

Las instrucciones están formadas por un código, un argumento inmediato opcional y una lista de argumentos extra opcionales dependiendo del código.

Se usa la siguiente notación para describir las instrucciones:

$$\texttt{codigo}[inmediato][, arg_1, \cdots, arg_n]$$

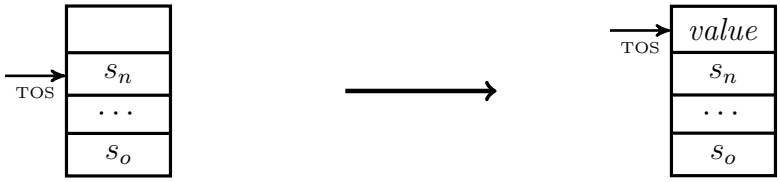
Se distingue entre los argumentos extra y el inmediato, ya que el tamaño de los argumentos extra es el doble del inmediato. Para distinguirlos, se utiliza el símbolo '`,`'.

[3]Del inglés: Top of stack

4.2.1 Conjunto de Instrucciones

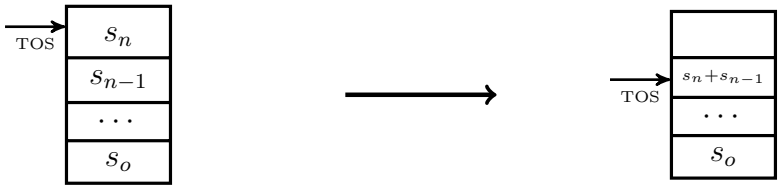
Instrucciones básicas

- `push` ,*value* La instrucción `push` coloca el valor *value* como tope del stack. En el diagrama a la izquierda se muestra el estado del stack antes de la operación y a la derecha el estado luego de su ejecución.



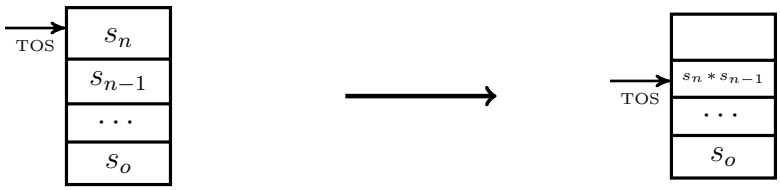
- `add`

  Remueve dos valores del stack, los suma y coloca el resultado en el tope.



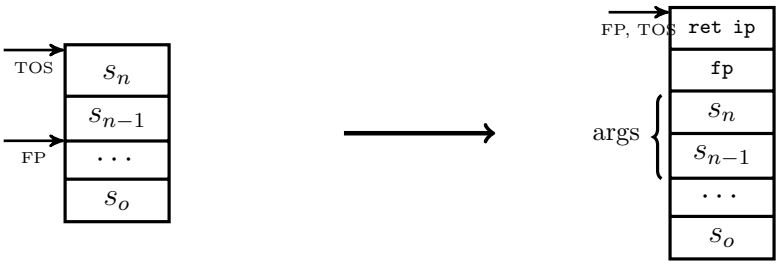
- `mul`

  Remueve dos valores del stack, los multiplica y coloca el resultado en el tope.



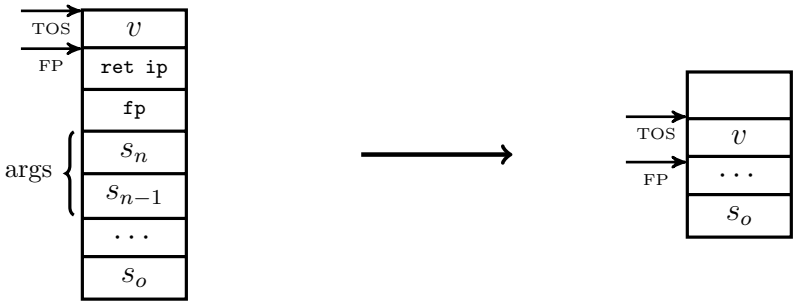
- `call`, arg_1

  Sirve para invocar una función, cuya posición en el código es arg_1. Coloca en el stack el valor de `fp` y coloca la posición siguiente en el código, donde seguirá ejecutando el código al retornar la función. Luego en `ip` coloca el valor arg_1.



- `ret`

  Sirve para retornar el valor resultado de la ejecución de una función. Toma del tope del stack el resultado `value`, revierte los valores de `sp` y `fp`, y coloca `value` en el nuevo tope del stack.

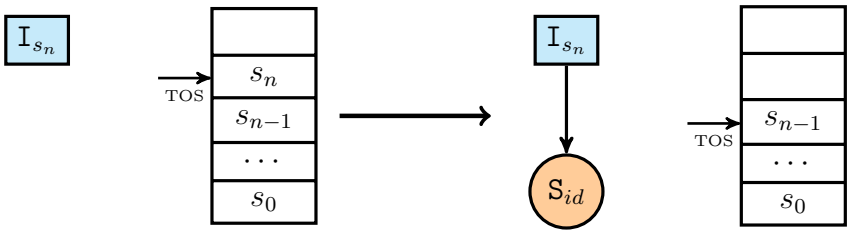


4.2.2 Instrucciones para manipular señales

A continuación se presentan las instrucciones utilizadas para manipular señales.

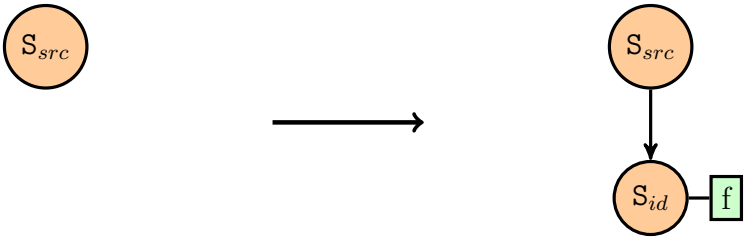
- `read` *id*

  Toma el tope del stack como identificador de una entrada. Crea una señal *id* que contendrá el valor de la entrada en el tiempo. Como precondición, la señal *id* no debe existir.



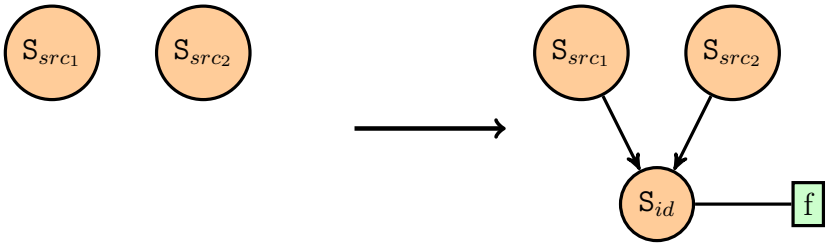
- `lift` *id*, *src f*

  Crea una señal *id* aplicando la función *f* a la señal *src*. Cada vez que la señal *src* cambie de valor, se le aplica la función *f* y la señal *id* cambia de valor.



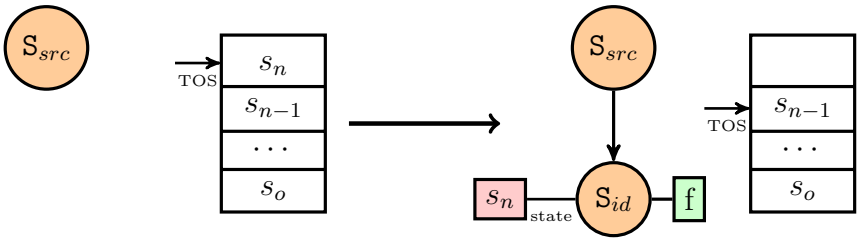
- `lift2` *id*, src_1 src_2 *f*

  Crea una señal *id* aplicando el combinador `lift2` usando la función *f*, y las señales src_1 y src_2. Cuando ambas señales cambien de valor, se aplica la función y la señal *id* cambia de valor.



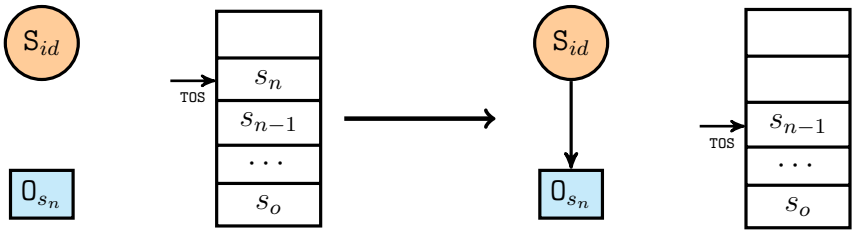
- `folds` *id*, *src f*

  Crea una señal *id* aplicando el combinador `folds`. El valor inicial de la señal está dado por el tope del stack, luego el mismo se actualiza aplicando la función *f* al valor actual y a los valores recibidos de la señal *src*.



- `write` *id*

  Envía los valores de la señal *id* a la salida identificada con el valor s_n ($\texttt{O}_{s_n}$) que se encuentra en el tope del stack (`TOS`).



En el Apéndice A.2.1 se encuentra el listado completo de operaciones y su descripción.

4.2.3 Ejemplo de traducción

Dado un robot con un sensor de distancia y un led, el siguiente programa Willie enciende el led cuando el robot detecta una casa.

```
#Inputs
INPUT_DISTANCE = 1
#Outputs
OUTPUT_LED = 1

isHouse distance = if (distance < 100) then 1 else 0

do {
  signal_distance <- read INPUT_DISTANCE
  signal_house <- lift isHouse signal_distance
  output OUTPUT_LED signal_house
}
```

El mismo se traduce a Alf de la siguiente forma:

```
0: call
1: 10
2: read 1
3: lift 0
4: 1
5: 16
6: call
7: 13
8: write 0
9: halt
10: push
11: 1
12: ret
13: push
14: 1
15: ret
16: load_param 0
17: push
```

```
18: 100
19: cmp_lt
20: jump_false
21: 26
22: push
23: 1
24: jump
25: 28
26: push
27: 0
28: ret
```

Para entender el programa Alf, primero se divide en dos secciones. Entre la línea 0 y la línea 9 está el código correspondiente a la sección `do`.

A partir de la línea 10 están las declaraciones de funciones.

La declaración:

```
10: push
11: 1
12: ret
```

se corresponde con la definición de la constante `INPUT_DISTANCE`, y la declaración

```
13: push
14: 1
15: ret
```

es la definición de la constance `OUTPUT_LED`.

La función `isHouse` se traduce a:

```
16: load_param 0
17: push
18: 100
19: cmp_lt
20: jump_false
21: 26
22: push
23: 1
24: jump
25: 28
26: push
27: 0
28: ret
```

En el bloque `do` la señal `signal_distance` se crea cargando el valor de `INPUT_DISTANCE` en la pila, y luego usando la instrucción `read`. El argumento 1 de la instrucción `read` será el identificador de la señal.

```
0: call
1: 10
2: read 1
```

Figura 4.3: Sistema de tipos en Willie

$$\frac{}{\Gamma \vdash () : \mathit{unit}} \text{ Unidad}$$

$$\frac{n \ \mathit{es\ un\ entero}}{\Gamma \vdash (n : \mathit{Number})} \text{ Numero}$$

$$\frac{\Gamma(v) = \mu}{\Gamma \vdash (v : \mu)} \text{ Variable}$$

$$\frac{\Gamma \vdash f : \mu \to \nu \quad \Gamma \vdash x : \mu}{\Gamma \vdash (fx : \nu)} \text{ Aplicacion}$$

$$\frac{\Gamma \vdash c : \nu \quad \Gamma \vdash a : \mu \quad \Gamma \vdash b : \mu}{\Gamma \vdash (\texttt{if}\ c\ \texttt{then}\ a\ \texttt{else}\ b : \mu)} \text{ Condicional}$$

$$\frac{\Gamma(i) = \tau}{\Gamma \vdash \texttt{input}i : \texttt{signal}\ \tau} \text{ Entrada}$$

$$\frac{\Gamma \vdash f : \mu_1 \to \cdots \to \mu_n \to \nu \quad \Gamma \vdash i_1 : \texttt{signal}\ \mu_1 \cdots \Gamma \vdash i_n : \texttt{signal}\ \mu_n}{\Gamma \vdash \texttt{lift}_n\ f\ i_1\ \cdots\ i_n : \texttt{signal}\ \nu} \text{ Lift}_\text{n}$$

$$\frac{\Gamma \vdash f : \mu \to \nu \to \nu \quad \Gamma \vdash c : \nu \quad \Gamma \vdash s : \texttt{signal}\ \mu}{\Gamma \vdash \texttt{folds}\ f\ c\ s : \texttt{signal}\ \nu} \text{ Folds}$$

$$\frac{\Gamma(o) = \tau \quad \Gamma \vdash s : \texttt{signal}\ \tau}{\Gamma \vdash \texttt{output}\ o\ s : \mathit{unit}} \text{ Salida}$$

Figura 4.4: Función de Fibonacci

```
# fibonacci
fibo n = if (n < 2) then 1 else fibo(n-1) + fibo(n-2)
```

Figura 4.5: Ejemplo completo

```
INPUT_DISTANCE = 1
OUTPUT_ENGINE = 1

MIN_DIST = 30
MAX_SPEED = 100
STOP = 0

distanceToSpeed n = if (n < MIN_DIST) then STOP else MAX_SPEED

do {
  distance <- read INPUT_DISTANCE,
  speed <- lift distanceToSpeed distance,
  output OUTPUT_ENGINE speed
}
```

Capítulo 5

Implementación

En este capítulo se detalla la implementación del compilador y la máquina virtual diseñadas para utilizar el lenguaje Willie en la plataforma elegida. También se explica cuál sería el mecanismo para portar la implementación a otra plataforma.

5.1 Compilador

El compilador será el encargado de leer el programa Willie y traducirlo a Alf.

El lenguaje utilizado para desarrollar el compilador fue *Haskell*. Las razones que llevaron a su elección son la portabilidad y la expresividad del mismo. El compilador `williec` es portable, ya que se puede compilar y ejecutar en diversos sistemas operativos utilizando el compilador *ghc*.

Es usual realizar tareas de compilación en *Haskell* por lo que existen herramientas estándar para cada etapa.

El compilador constará de una secuencia de etapas: Análisis Léxico, Análisis Sintáctico, Análisis Semántico y Generación de Código.

En la Figura 5.1 se puede ver la estructura más detallada del compilador y a continuación se describe cada etapa representada en la Figura.

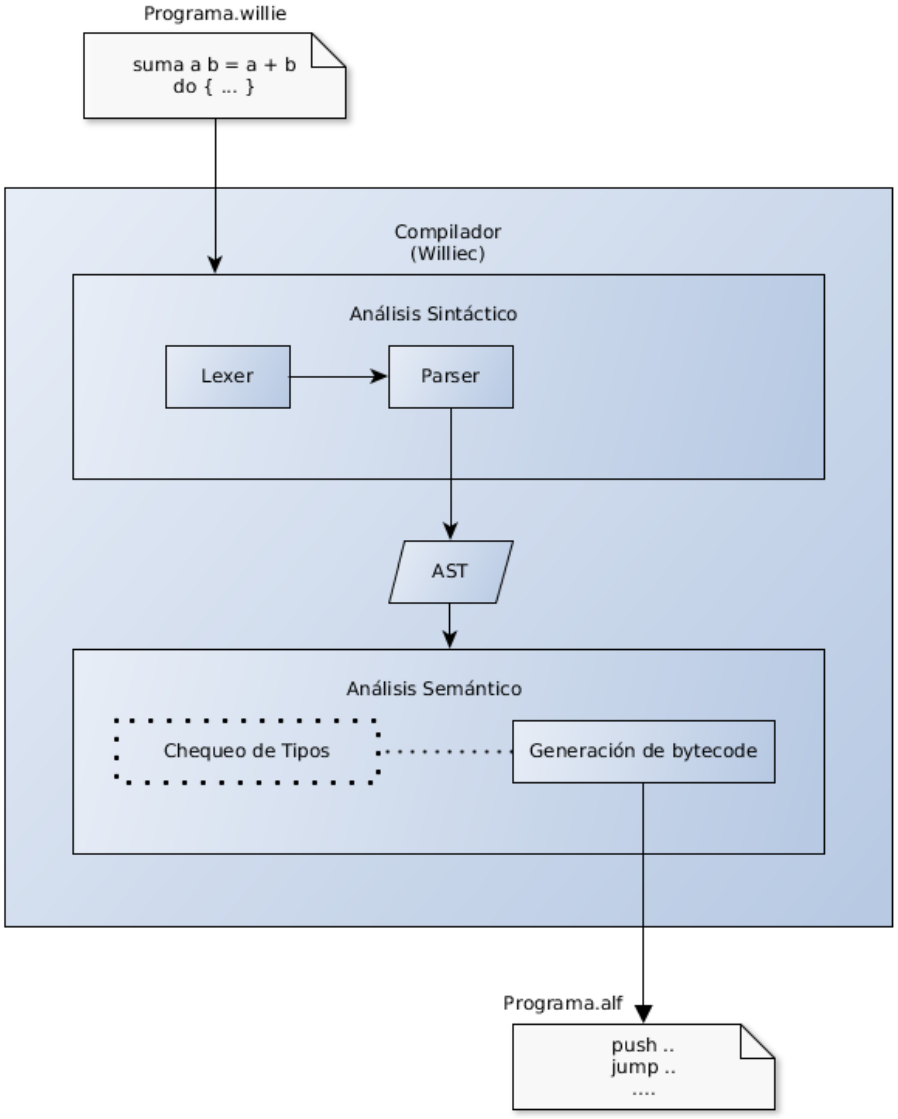
5.1.1 Análisis Léxico

La primera etapa se llama análisis léxico, en esta se lee el código fuente en lenguaje Willie (.willie) y se transforma a una lista de lexemas.

Un lexema puede ser una palabra reservada (ej: `do`), un valor (ej: 19), un identificador (ej: `distance`) o un símbolo reservado (ej: `+`).

Para representar los lexemas, se utiliza la herramienta *UU.Scanner* [25] que estandariza los mismos en el tipo de datos `Token`.

Figura 5.1: Diagrama del compilador



Usando *Alex*[26] se procesa el código fuente, se reconocen los lexemas y se retorna una lista de tipo `[Token]`.

La etapa se puede resumir en la implementación de la función `tokenize`.

```
tokenize :: String -> String -> [Token]
```

5.1.2 Análisis Sintáctico

La segunda fase del compilador, recibe la lista de lexemas (`[Token]`) y reconoce el lenguaje, generando un árbol de sintaxis abstracta (AST[1]).

Para reconocer la gramática se implementó un parser recursivo descendente. Utilizando la herramienta *UU.Parser* [25], se definió un tipo de datos `TokenParser a` que representa un parser que recibe una secuencia de lexemas de tipo `Token` y retorna un AST de tipo `a`.

```
type TokenParser a = Parser Token a
```

UU.Parser define un conjunto de combinadores de parsers y utilizándolos se construyen parsers complejos a partir de parsers simples.

Para representar el AST se utiliza una gramática de atributos. Una gramática de atributos es como una gramática libre de contexto, pero agrega semántica a la misma. Para el análisis sintáctico, la semántica no es utilizada, pero será usada en la próxima etapa.

El sistema de gramáticas de atributos $UUAG$[27] fue usado para la implementación.

Se define un tipo de datos `Root` que representa la raíz del árbol.

```
data Root
  | Root
    decls :: Decls
    dodecls :: Dodecls
```

El mismo tiene un único constructor `Root_Root` que recibe un árbol de tipo `Decls` que representa las declaraciones, y un árbol de tipo `Dodecls` que representa el bloque `do`.

Para crear el AST usando *UU.Parser* se define el parser `pRoot`:

```
pRoot :: TokenParser Root
pRoot
  = Root_Root <$> pDecls <*> pDodecls
```

El cuál asume definido un parser de declaraciones `pDecls` y un parser del bloque `do` (`pDodecls`).

```
pDecls :: TokenParser Decls

pDodecls :: TokenParser Dodecls
```

[1] Del inglés Abstract Syntax Tree

El combinador "$\langle * \rangle$" [28] se utiliza para combinar dos parser y resolver producciones de largo 2 en una gramática, en este caso reconocer primero la lista de declaraciones de funciones, y luego el bloque `do`. El tipo del combinador es:

$$(\langle * \rangle) :: \texttt{Parser}\ s\ a \rightarrow \texttt{Parser}\ s\ b \rightarrow \texttt{Parser}\ s\ (a, b)$$

El combinador "$\langle \$ \rangle$" se utiliza para aplicar una función al resultado de un parser, en éste caso la función es aplicar el constructor `Root_Root`.

Para construir el parser completo, se va refinando sucesivamente en parsers mas específicos, hasta construir completamente el *AST*. En el apéndice A.3 se encuentra código del parser implementado.

5.1.3 Análisis Semántico

Para la última etapa se utiliza la gramática de atributos para definir semántica sobre el *AST*.

Las gramáticas de atributos (*Attribute Grammars* [29] [27]) simplifican la tarea de escribir catamorfismos. Un catamorfismo es una función análoga a la función de alto orden `foldr` pero aplicada sobre cualquier tipo de datos recursivo.

De ésta manera se pueden definir atributos sintetizados, heredados o mixtos en el *AST*.

Los atributos sintetizados son valores que se distribuyen desde las hojas hacia la raiz del *AST*, y los heredados aquellos que se distribuyen desde la raiz hacia las hojas.

Uno de dichos atributos será el código en bajo nivel, la salida de esta etapa.

Se implementó una gramática de atributos usando *UUAG*, y con el compilador de gramáticas *UUAGC*[30] se compiló a `Haskell`.

El compilador *UUAGC* toma la gramática y construye los catamorfismos necesarios para procesar todos los atributos.

Construir el compilador, se reduce a obtener una secuencia de atributos sobre el *AST* que sirven para generar el código Alf.

Por ejemplo para construir el código de un programa, la raiz del *AST* está dada por el tipo de datos `Root`.

```
data Root
  | Root
      decls :: Decls
      dodecls :: Dodecls
```

Se puede definir el código como la concatenación del código de las declaraciones del bloque `do`, una instrucción `halt` y el código de las declaraciones de funciones (`Decls`).

Para ésto se define un atributo sintetizado (`syn`) llamado `code`.

```
set All = Root Decls Decl Dodecls Dodecl Expr
attr All syn code use {++} {[]} :: BC

sem Root
  | Root
    lhs.code = @dodecls.code ++ [Thalt] ++ @decls.code
```

Utilizando la palabra clave `Set` se define el conjunto `All` de elementos para los que se definirá el atributo.

En la definición del atributo, se especifica que en caso de no haber una regla específica, se calcula usando la concatenación `++` y como atributo por defecto toma `[]`. A ésto se le llama *use rule*.

```
  attr All syn code use {++} {[]} :: BC
```

Por ejemplo para la definición de la lista de declaraciones:

```
type Decls = [Decl]
```

No es necesario especificar que el código se obtiene concatenando sus partes ya que se infiere automáticamente usando la regla anterior.

Para poder generar el código de todo el programa, es necesario calcular otros atributos previos. Se necesita saber la posición en la que quedarán las funciones para poder tener una referencia a ellas. Para saber la posición, es necesario calcular el largo del código antes de tener el código.

Para ésto se definió un atributo sintetizado `len` que contiene el largo que tendrá cada bloque luego de traducido a código, pero sin llegar a traducirlo.

También se definió un atributo `pos` que indica en que posición estará ubicado el código que se genere para cada producción de la gramática. El atributo `pos` es un atributo heredado (`inh`) en el *AST*.

Por ejemplo en `Root`, se utiliza el atributo `len` de las declaraciones del bloque `do` para saber a partir de que posición `pos` estarán ubicadas las declaraciones de funciones.

```
attr All syn code use {++} {[]} :: BC
         syn len use {+} {0} :: Int
         inh pos :: Int
```

```
sem Root
  | Root
      lhs.code = @dodecls.code ++ [Thalt] ++ @decls.code
      dodecls.pos = 0
      decls.pos = @dodecls.len + 1
```

Utilizando el atributo `pos`, se puede saber en que posición estará cada función en el código generado. Para tener la posición de todas las funciones se utiliza un atributo encadenado (`chn`) llamado `labels`, es heredado pero también es sintetizado. Por ejemplo al declarar una función, se agrega la posición `pos` asociada al nombre de la misma.

```
sem Decl
  | Function
      lhs.code = @body.code ++ [Tret]
      lhs.len = @body.len + 1
      lhs.labels = addLabel @name @lhs.pos @lhs.labels
```

El atributo contiene un mapa que dado un nombre de una función retorna la posición de la misma, `labels` recolecta la posición de todas las funciones. Luego otro atributo `labelMap` se declara como heredado `inh` y se le asigna en `Root` el valor de `labels`, `labelMap` se usa para distribuir el mapa completo en todo el *AST*.

```
sem Root
  | Root
      decls.labels = emptyLabelMap
      decls.labelMap = @decls.labels
      dodecls.labelMap = @decls.labels
```

Por último un atributo `env` encadenado recolecta las declaraciones de identificadores, a cada identificador de señal le asigna un número entero único y mantiene la lista de las variables en el alcance (scope) dentro de una función. Luego que el atributo `env` recolecta todas las declaraciones, el resultado es distribuido con el atributo heredado `envInh`.

```
sem Root
  | Root
      decls.env = emptyEnv
      dodecls.env = emptyEnv
      dodecls.envInh = @dodecls.env
```

Usando todos éstos atributos se genera el código para cada producción de la gramática, y el atributo `code` se puede calcular.

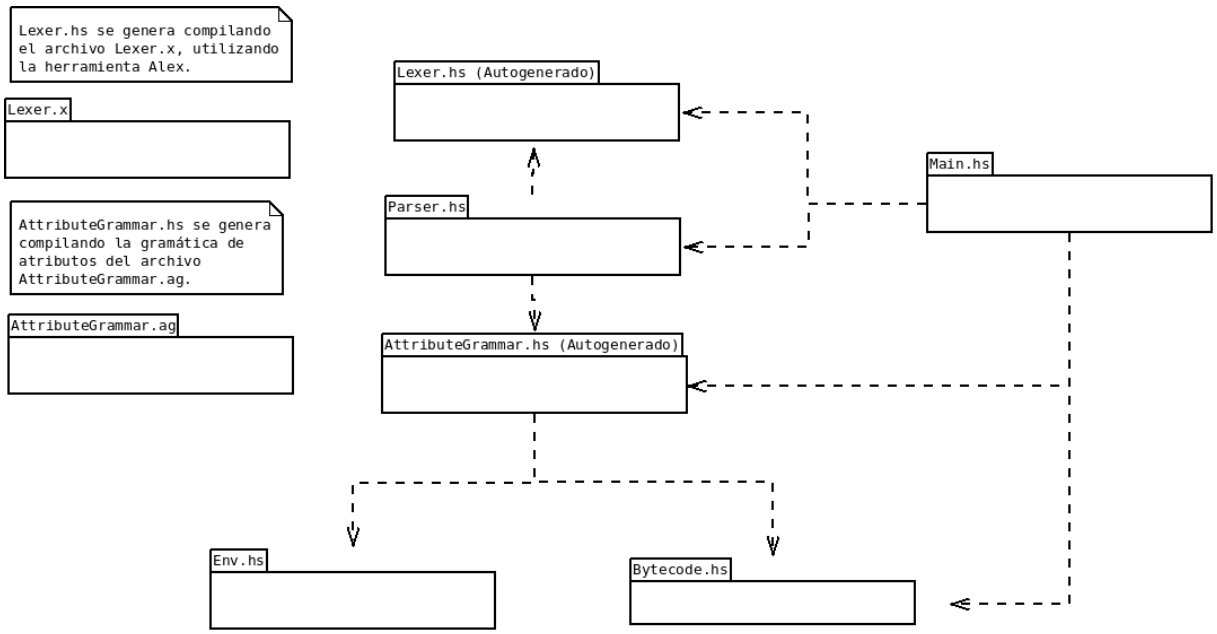
Se definió un módulo `Bytecode` que abstrae el código de máquina en un tipo `OpCode` y define funciones para exportarlo como texto o en formato binario.

Al compilar la gramática usando *UUAGC* se obtiene un módulo en lenguaje `Haskell` que expone la función `code_Syn_Root` y deja accesible el código resultado como una lista de tipo `[OpCode]`.

Utilizando el módulo `Bytecode`, el código se obtiene y escribe en un archivo (.alf) terminando el proceso de compilación.

Los módulos del compilador y sus dependencias se pueden ver en la Figura 5.2.

Figura 5.2: Paquetes del compilador `williec`



5.2 Máquina virtual

La máquina, deberá ejecutar el código de bajo nivel en una 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 [31] y en [32].

El lenguaje de programación elegido para el desarrollo de la máquina virtual es *C++* ya que es posible compilarlo para casi cualquier plataforma objetivo. Además *C++* permite acceder a muy bajo nivel, y manipular a nivel de *bytes* las estructuras.

MBED es una plataforma pensada para colaborar mediante un entorno de desarrollo web, y compilador online, ese esquema de trabajo no es el más práctico para desarrollar la máquina virtual, por lo que se descargaron de la página de mbed [33], las herramientas de desarrollo para compilar offline.

La máquina tendrá dos partes principales, una que interpreta el código e implementa el despachador que actualiza las señales. Esta parte es común y puede ser portada a diferentes plataformas sin necesidad de modificarla.

Las instrucciones en memoria tendrán un ancho de palabra de 16 bit. La primera palabra contiene en los 8 bits más representativos, el código de la operación (`opcode`). Los 8 bits menos representativos, contienen un argumento inmediato opcional. Luego según el `opcode`, algunas instrucciones pueden tener argumentos adicionales, en las siguientes palabras de 16 bit.

codigo de 8 bit	inmediato de 8 bit
argumento 1 opcional de 16 bit	
...	
argumento n opcional de 16 bit	

La máquina mantiene un puntero a la siguiente instrucción a ejecutar llamado `ip` [2]. Cuando `ip` no es nulo, la máquina ejecuta todas las instrucciones hasta que el mismo se haga nulo. El pseudocódigo de la máquina es:

```
1 - Crear grafo de señales vacío, inicializar pila.
2 - Apuntar ip al inicio del código.
3 - Ejecutar código hasta que ip se haga nulo.
```

[2]IP: del inglés, Instruction Pointer significa puntero a instrucción

```
4 - Para siempre:
  4.1 - Leer entradas
  4.2 - Actualizar señales conectadas a las mismas, marcar como listas.
  4.2 - Mientras hay señales listas para procesar:
    4.2.1 - Cargar valores en stack.
    4.2.2 - Apuntar ip a inicio de la función asociada.
    4.2.3 - Ejecutar hasta que ip se haga nulo.
    4.2.4 - Actualiza señales conectadas a la misma.
  4.3 - Escribir salidas.
```

Al principio se crea un grafo de señales vacío, y se reserva espacio para la pila, todo en memoria estática, tanto para los nodos del grafo como para la pila. En el punto 2, se interpretan las instrucciones del programa, hasta llegar a la instrucción `halt`. Las instrucciones al inicio del código se corresponden con el bloque `do` del programa, por lo tanto al ejecutarlo se obtiene el grafo de las señales completo.

Con el grafo armado, luego se obtienen los valores de las entradas necesarias, y para cada señal conectada se calcula su valor.

Luego, en el punto 4.2 el grafo de señales se recorre en orden topológico, actualizando el valor de cada señal.

Es sencillo notar, que la actualización de las señales listas para procesar es un punto que con pocas modificaciones, puede ser realizado en paralelo en un entorno multiprogramado.

Las señales que estén conectadas a una salida, se usan para actualizar el valor de las mismas.

Para cada instrucción hay una función definida que interpreta, el despachador tomará una a una las instrucciones e invocará la función que la maneja de acuerdo a que operación es. Las funciones son de tipo `void` y realizan cambios sobre el estado de la máquina. La referencia a las mismas es guardada en un vector `functions`. La posición de cada instrucción en el vector coincide con el código de operación.

```
void (*functions[])() = {
    f_halt,
    f_call, f_ret, f_load_param,
    f_lift, f_lift2, f_folds,
    f_read, f_write,
    f_jump, f_jump_false,
    f_cmp_eq, f_cmp_neq, f_cmp_gt, f_cmp_lt,
    f_add, f_sub, f_div,
    f_mul, f_op_and, f_op_or,
```

```
    f_op_not,
    f_push, f_pop, f_dup,
    f_store, f_load
};
```

Por ejemplo, al despachar el operador `push`, la siguiente palabra contiene el valor a colocar en el stack. El código en lenguaje `C` que maneja la instrucción es:

```
void f_push() {
    *++sp = *ip++;
}
```

Se creó un archivo `Makefile` para construir una imagen binaria de la máquina virtual. Ésto genera un archivo *mbed_alfvm.bin*. Para cargar la máquina en la placa *MBED*, alcanza con conectarla a un puerto USB y pegar el archivo en la carpeta /media/MBED.

Capítulo 6

Caso de estudio

En esta sección veremos un caso de estudio usado para verificar la implementación. El problema fue tomado de la competencia SumoUY [4], el mismo fue el desafío planteado a escolares en el año 2013.

6.1 Problema

"*Se desea implementar un robot autónomo móvil que sea capaz de hacer la entrega de un pedido en una casa determinada. El mismo debe moverse por un escenario e identificar las casas. Para recorrer la ruta de entrega, podrá valerse de una línea negra que representará la calle de la ciudad.*

Las casas estarán ubicadas a un lado de la calle. En el recorrido se encuentran varias casas, el robot deberá entregar un pedido en la quinta casa por la que pase.

El robot deberá pasar por alto las casas anteriores y al llegar a la casa objetivo debe detenerse totalmente.

Para probar la solución, se armará un escenario que consiste de un piso blanco con una línea negra que puede tener curvas.

Al lado derecho de la línea se ubicarán cajas a menos de 30 centímetros representando las casas."

6.2 Solución

Se armó un robot móvil que cuenta con 3 sensores:

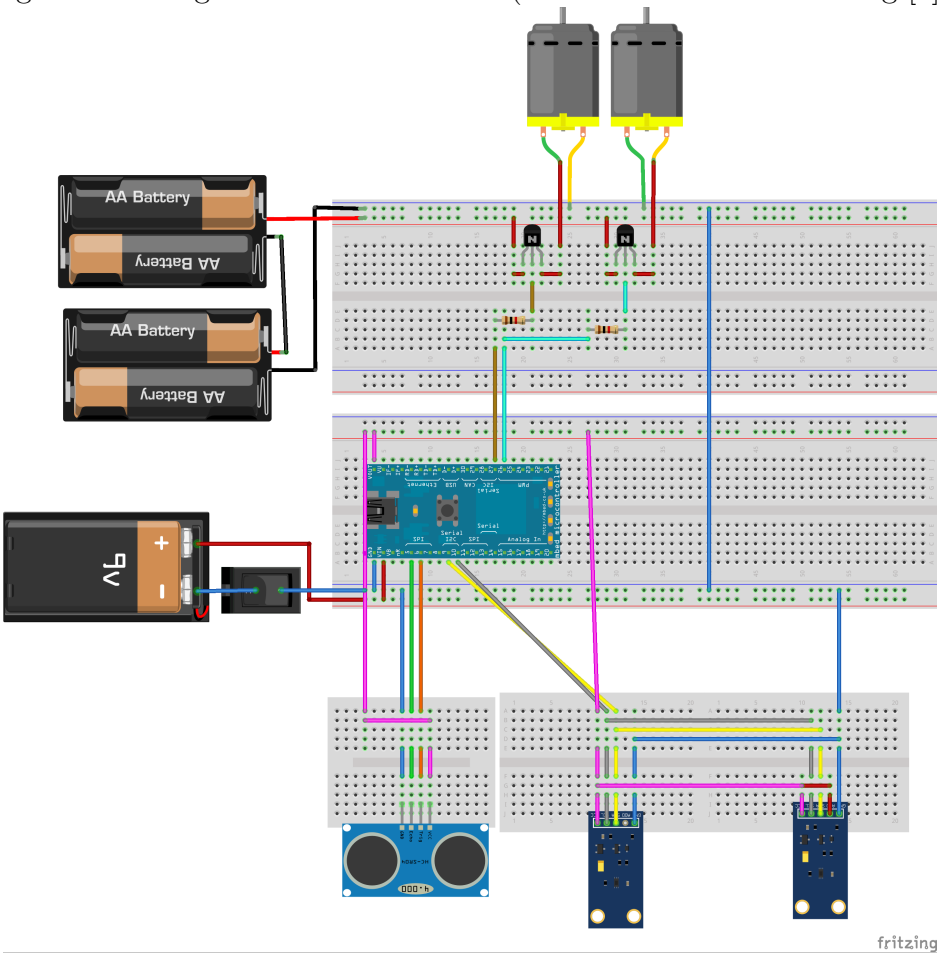
- Sensor de grises izquierdo
- Sensor de grises derecho

- Sensor de distancia apuntando hacia la derecha

Y 2 actuadores:

- Motor izquierdo
- Motor derecho

Figura 6.1: Diagrama del robot móvil (realizado utilizando fritzing [3])



El diagrama de la Figura 6.1 muestra los componentes físicos que son montados en el robot para resolver el problema y cómo se interconectan. Arriba se pueden ver los dos motores, que irán uno a cada lado del robot y

sólo se moverán hacia adelante. Se utilizan salidas *pwm* [1] del MBED para controlar la velocidad de cada motor.

Los motores necesitan más energía que la que se puede entregar con los pines de salida del MBED, y para ésto tienen su propia fuente de voltaje. Se utilizan dos transistores para amplificar la señal que controla cada motor.

El robot utilizará dos sensores de grises montados al frente para mantenerse sobre la línea, ambos pueden verse a la derecha abajo en la Figura 6.1. Con los motores el robot se moverá hacia adelante inicialmente, e irá corrigiendo su dirección desacelerando el motor del lado que se salga de la línea. Junto a cada sensor de grises se montará una luz led, que de acuerdo al color del suelo, se reflejará y se podrá decidir si se está viendo algo oscuro (la línea) o algo claro (fuera de la línea).

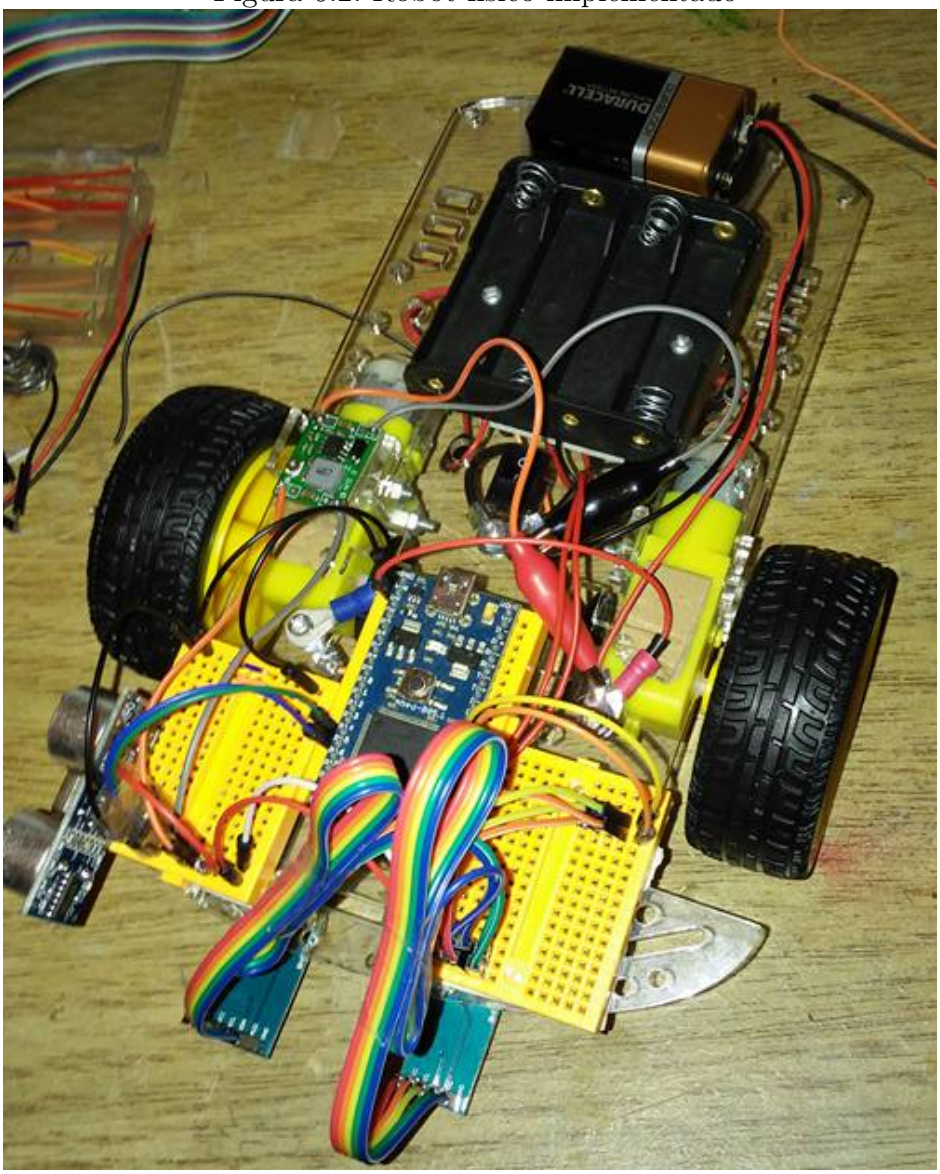
El sensor de distancia a la izquierda debajo en la figura, se montará en el robot apuntando hacia la derecha, para saber cuándo el mismo está pasando frente a una casa.

Durante el trayecto se mantendrá la cuenta de las casas, y el robot se detendrá totalmente cuando la cuenta llegue al valor 5.

En la Figura 6.2 se puede ver el robot físico creado como prototipo para probar el caso de estudio.

[1] PWM: Del inglés, pulse width modulation; Modulación por ancho de pulsos. Se utiliza para crear señales de voltaje en ciclos periódicos y controlar la cantidad de energía que se envía.

Figura 6.2: Robot físico implementado



6.2.1 Implementación usando Willie

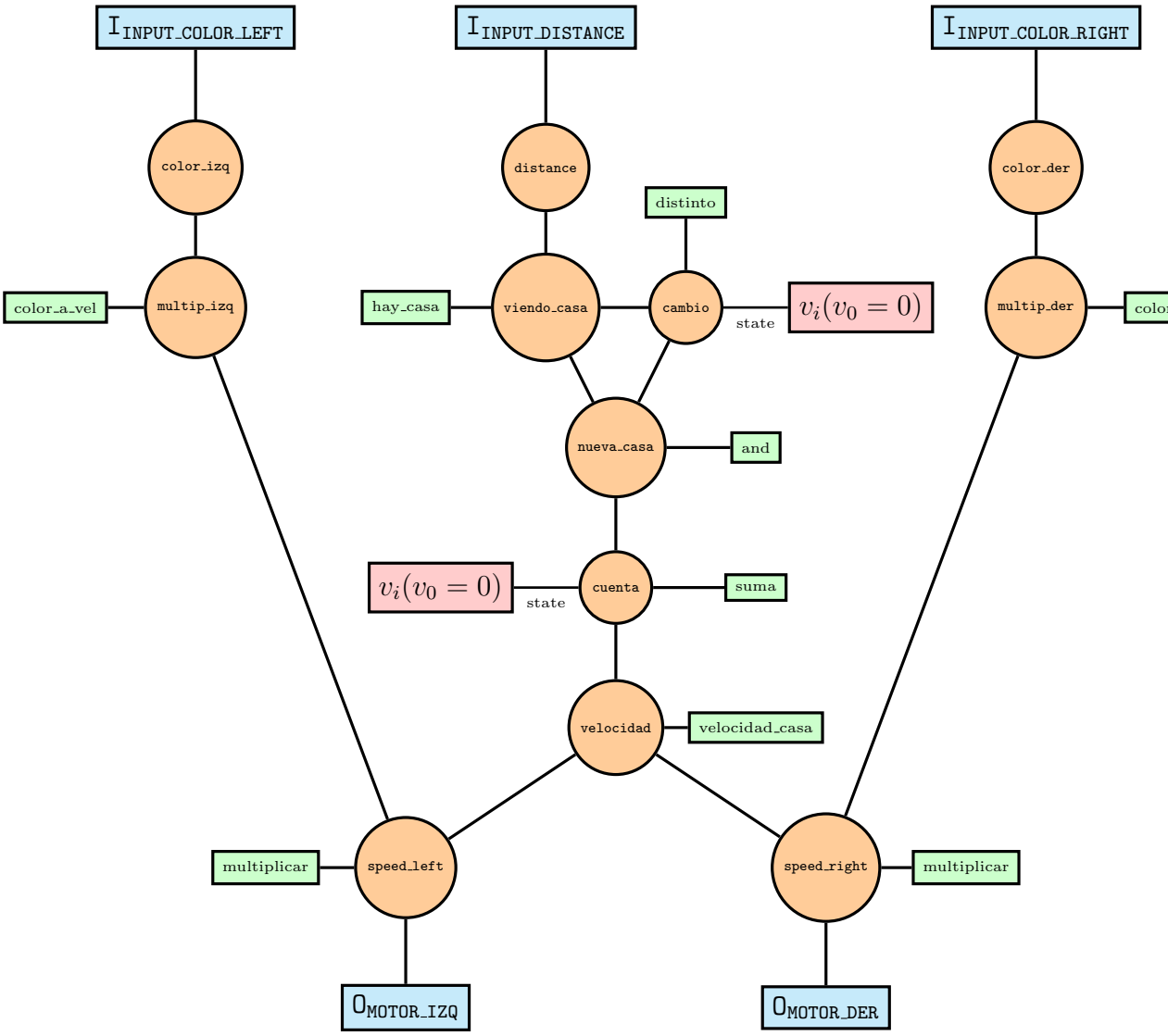
Luego se llega a la implementación en el lenguaje Willie:

```
INPUT_DISTANCE = 1
INPUT_COLOR_LEFT = 2
INPUT_COLOR_RIGHT = 3
OUTPUT_ENGINE_LEFT = 1
```

```
OUTPUT_ENGINE_RIGHT = 2

MIN_DISTANCE = 100
MIN_GREY = 50

hay_casa d = if (d < MIN_DISTANCE) then 1 else 0
distinto a b = if (a /= b) then 1 else 0
velocidad_casa num = if (num >= 5) then 0 else 100

and a b = if (a && b) then 1 else 0
suma a b = (a + b)
multiplicar a b = (a * b)

color_a_vel gris = if (gris > MIN_GREY) 1 else 1/2


do {
    distance <- read INPUT_DISTANCE,
    color_izq <- read INPUT_COLOR_LEFT,
    color_der <- read INPUT_COLOR_RIGHT,

    viendo_casa <- lift hay_casa distance,
    cambio <- folds distinto 0 viendo_casa,
    nueva_casa <- lift2 and viendo_casa cambio,
    cuenta <- folds suma 0 nueva_casa,
    velocidad <- lift velocidad_casa cuenta,

    multip_izq <- lift color_a_vel color_izq,
    multip_der <- lift color_a_vel color_der,

    speed_left <- lift2 multiplicar velocidad multip_izq,
    speed_right <- lift2 multiplicar velocidad multip_der,

    output MOTOR_IZQ speed_left,
    output MOTOR_DER speed_right
}
```

6.2.2 Diagrama de la solución

Utilizando la notación definida en la Sección 4.2, en la Figura **??** se puede ver gráficamente de qué forma se combinan los eventos para lograr el objetivo.



6.3 Conclusiones del caso

A diferencia de un programa imperativo tradicional, en el programa Willie dentro del bloque `do` se puede ver claramente cómo se procesan las entradas, para generar los valores de las salidas. Al razonar utilizando funciones sobre

señales, no es necesario tener en cuenta estado global, ni el orden de ejecución de diferentes rutinas.

Al estar obligado a escribir funciones puras, el desarrollador puede abstraerse mejor al pensar que funciones necesita implementar, y que entradas tomarán. En un programa imperativo, es normal realizar operaciones de entrada y salida dentro de cualquier función, lo que dificulta ver cuándo hay efectos secundarios de invocar cada función.

Finalmente, no es necesario preocuparse por la concurrencia, sino por la semántica del programa, la máquina virtual se encargará de respetar las definiciones del desarrollador.

Tampoco es necesario preocuparse por las interacciones de entrada y salida, ya que por decisión de diseño existen abstracciones en la máquina bien definidas para cada una.

Si bien Willie es muy diferente a los lenguajes de programación imperativos tradicionales y ésto puede ocasionar una curva inicial de aprendizaje pronunciada, finalmente puede enseñar al desarrollador una forma diferente de razonar y pensar la solución a los problemas de robótica.

El problema se pudo representar en la plataforma `MBED LPC1768` antes mencionada, utilizando el robot construido. La memoria Flash total utilizada en la prueba sumó 44 KB, por lo que aún faltaría mejorar la implementación para llegar a menos de 32 KB y cubrir mas plataformas. Sin embargo el programa Willie compilado a Alf ocupa tan solo 88 Bytes, el hecho de que se utilicen abstracciones para entrada y salida, ayuda a mantener bajo el tamaño de los programas Alf. El resto es ocupado por la implementación de la máquina virtual. La memoria RAM utilizada, al usar memoria estática no supera los 512 Bytes, y no crece con el correr del tiempo.

Éstos resultados indican que es posible realizar desafíos robóticos y se puede escalar en tamaño, quedando como necesidad, reducir aún más el tamaño de la máquina virtual.

Capítulo 7

Conclusiones

Se diseñaron los lenguajes Willie y Alf que podrán ser retomados en futuros trabajos para ser mejorados y extender su funcionalidad, eliminando las restricciones impuestas por el alcance actual.

Se realizó una implementación modelo de cada parte del diseño necesaria para poder escribir programas en alto nivel, usando el paradigma de programación funcional reactiva, y se logró seguir todos los pasos necesarios para ejecutar dichos programas en plataformas con bajas capacidades de cómputo.

A diferencia de otros trabajos, en éste proyecto los robots resultantes son autónomos y no requieren de contacto con el exterior ni directivas para realizar sus tareas.

Es posible escribir programas cortos, en alto nivel, que permitan pensar en los problemas y sus soluciones, sin necesidad de detenerse a ver cada parte a bajo nivel.

Se logró recortar el alcance de cada parte del proyecto para poder ser implementado por una sola persona.

La elección del lenguaje *Haskell* para implementar el compilador fue correcta, ya que fácilmente se pueden realizar modificaciones, algo que durante el proyecto agregó mucho valor.

Se implementó la máquina virtual para la plataforma *MBED* y el tamaño de la misma, así como la capacidad de procesamiento no fue un problema, al utilizar `C++` como lenguaje y controlar el uso de la memoria y las estructuras usadas. Lamentablemente dado el alcance, no se pudo hacer pruebas con plataformas aún mas pequeñas como `Arduino` y `PIC`.

Se construyó un robot físico, que permitió completar el flujo de trabajo y ver como resultado final, la ejecución del caso de estudio, utilizando todas las herramientas desarrolladas.

A pesar de las limitaciones impuestas por el alcance, sería posible luego de algunas mejoras utilizar el lenguaje para enseñar conceptos de robótica.

7.1 Trabajo futuro

Como principal trabajo futuro, se notó la necesidad de contar con un simulador, para eliminar la necesidad de construir un robot físico para resolver un problema. Permitiría también crear diversos escenarios y tener una mayor diversidad de ambientes de prueba, para luego crear el robot físico deseado e interactuar con el mismo.

También sería muy útil contar con una funcionalidad de depuración, la cual mostrara dependiendo del tiempo los valores de cada entrada, cada señal y cada salida.

Una posible opción de implementar depuración es comunicar mediante el puerto serial el valor de cada señal al cambiar, y mostrarlo en una interfaz web como la que provee RXMarbles (ver [34]).

El lenguaje *Elm* provee de una herramienta que permite "viajar en el tiempo", no solo permite ver los valores de las señales sino también modificarlos y seguir la ejecución de un programa. En nuestro caso no sería posible modificar lo que el robot físico realiza, pero si sería útil ver en la línea de tiempo que valores tomaron sus señales. (ver [35]) Además, si se contara con un simulador sería posible.

Por el alcance del proyecto no se incluyó el chequeo del sistema de tipos en la implementación del compilador.

Bibliografía

[1] Sitio web de Yampa. `https://wiki.haskell.org/Yampa`. Último acceso: 29/04/2015.

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

[3] Sitio web del fritzing. `http://fritzing.org/`. Último acceso: 13/08/2015.

[4] Sitio web de SumoUY. `http://sumo.uy/`. Último acceso: 03/05/2015.

[5] Conal Elliott and Paul Hudak. Functional reactive animation. In *International Conference on Functional Programming*, 1997.

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

[7] John Peterson, Gregory D. Hager, and Paul Hudak. A language for declarative robotic programming. In *IEEE International Conference on Robotics and Automation*, 1999.

[8] Zhanyong Wan, Walid Taha, and Paul Hudak. Real-time FRP. In *International Conference on Functional Programming*, 2001.

[9] Zhangyong Wan, Walid Taha, and Paul Hudak. Event-driven FRP. jan 2002.

[10] Sitio web de Microchip. `https://www.microchip.com`. Último acceso: 13/08/2015.

[11] Henrik Nilsson, Antony Courtney, and John Peterson. Functional reactive programming, continued. oct 2002.

[12] Paul Hudak, Antony Courtney, Henrik Nilsson, and John Peterson. Arrows, robots, and functional reactive programming. aug 2002.

[13] John Hughes. Generalising monads to arrows. In *Science of Computer Programming*, 1998.

[14] Sitio web de Arduino. `https://www.arduino.cc`. Último acceso: 23/11/2015.

[15] Sitio web de GNU Compiler Collection. `https://gcc.gnu.org`. Último acceso: 23/11/2015.

[16] Sitio web de Atmel. `https://www.atmel.com`. Último acceso: 23/11/2015.

[17] Sitio web de Atmel. `https://www.atmel.com`. Último acceso: 23/11/2015.

[18] Sitio web de ARM. `https://www.arm.com`. Último acceso: 23/11/2015.

[19] Sitio web del proyecto Butia. `https://www.fing.edu.uy/inco/proyectos/butia/`. Último acceso: 23/11/2015.

[20] Sitio web de PIC 18F4550. `https://www.microchip.com/PIC18F4550`. Último acceso: 23/11/2015.

[21] Sitio web de Tortubots. `https://www.fing.edu.uy/inco/proyectos/butia/mediawiki/index.php/TortuBots`. Último acceso: 23/11/2015.

[22] Sitio web de Butialo. `https://www.fing.edu.uy/inco/proyectos/butia/mediawiki/index.php/Butialo`. Último acceso: 23/11/2015.

[23] Sitio web del proyecto Yatay. `https://www.fing.edu.uy/inco/proyectos/butia/mediawiki/index.php/Yatay`. Último acceso: 23/11/2015.

[24] Sitio web de Beagleboard Black. `http://beagleboard.org/BLACK`. Último acceso: 06/12/2015.

[25] Sitio web de Ulrecht University Parser Combinators (uu-parsinglib). `http://foswiki.cs.uu.nl/foswiki/HUT/ParserCombinators`. Último acceso: 25/10/2015.

[26] Sitio web de Alex. `http://www.haskell.org/alex`. Último acceso: 26/07/2015.

[27] Sitio web de Ulrecht University Attribute Grammar System (UUAG). `http://foswiki.cs.uu.nl/foswiki/HUT/AttributeGrammarSystem`. Último acceso: 25/10/2015.

[28] S. Doaitse Swierstra. Combinator parsers: a short tutorial. In A. Bove, L. Barbosa, A. Pardo, and J. Sousa Pinto, editors, *Language Engineering and Rigorous Software Development*, volume 5520 of *LNCS*, pages 252–300. Spinger, 2009.

[29] Sitio web de Attribute Grammars. `http://wiki.haskell.org/Attribute_grammar`. Último acceso: 29/11/2015.

[30] Sitio web de Ulrecht University Attribute Grammar System Compiler (UUAGC). `http://foswiki.cs.uu.nl/foswiki/HUT/AttributeGrammarSystem`. Último acceso: 25/10/2015.

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

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

[33] Sitio web Exporting Mbed to GCC ARM. `https://developer.mbed.org/handbook/Exporting-to-GCC-ARM-Embedded`. Último acceso: 26/07/2015.

[34] Sitio web de RXMarbles. `http://rxmarbles.com`. Último acceso: 18/05/2015.

[35] Sitio web del debugger de Elm. `http://debug.elm-lang.org/`. Último acceso: 18/05/2015.

Apéndices

Apéndice A

Apéndice

A.1 Manual de usuario

Para utilizar el compilador, dado un archivo *Ejemplo.willie*, se ejecuta:

```
> williec < Ejemplo.willie > Ejemplo.alf
```

El código de la máquina virtual está en el directorio /src/alfvm, para compilarlo se ejecuta:

```
> cd src/alfvm
> make
```

A.2 Manual de Referencia

A.2.1 Instrucciones de bajo nivel

A continuación se presenta el resto de las instrucciones de bajo nivel y pseudocódigo indicando su semántica.

- `halt`

  Detiene el hilo de ejecución actual.

  `ip = 0`

- `call function`

  Invoca la funcion *function*. Se asume que los parámetros están en el stack.

- `ret`

  Toma el resultado de una función del tope del stack, limpia el espacio ocupado por la función, y deja el resultado en el nuevo tope del stack.

  ```
  value = stack.pop()
  stack.pop_frame();
  ip = code[stack.pop()]
  fp = stack.pop()
  stack.pop_args()
  stack.push(value)
  ```

- `load_param inm`

  ```
  a = stack.get_arg(inm)
  stack.push(a)
  ```

- `jump`

  ```
  goto position
  ```

- `jump_false position`

  ```
  a = stack.pop()
  if not a: goto position
  ```

- `cmp_eq`

  ```
  a = stack.pop()
  b = stack.pop()
  stack.push(a == b)
  ```

- `cmp_neq`

  ```
  a = stack.pop()
  b = stack.pop()
  stack.push(a != b)
  ```

- `cmp_gt`

  ```
  a = stack.pop()
  b = stack.pop()
  stack.push(a >b)
  ```

- cmp_lt

  ```
  a = stack.pop()
  b = stack.pop()
  stack.push(a <b)
  ```

- add

  ```
  a = stack.pop()
  b = stack.pop()
  stack.push(a + b)
  ```

- sub

  ```
  a = stack.pop()
  b = stack.pop()
  stack.push(a - b)
  ```

- div

  ```
  a = stack.pop()
  b = stack.pop()
  stack.push(a / b)
  ```

- mul

  ```
  a = stack.pop()
  b = stack.pop()
  stack.push(a * b)
  ```

- op_and

  ```
  a = stack.pop()
  b = stack.pop()
  stack.push(a and b)
  ```

- op_or

  ```
  a = stack.pop()
  b = stack.pop()
  stack.push(a or b)
  ```

- `op_not`

  Coloca un valor constante en el stack.

  `stack.push(word)`

- `push word`

  Coloca un valor constante en el stack.

  `stack.push(word)`

- `pop`

  Elimina el tope del stack.

  `stack.pop()`

- `dup`

  Duplica el tope del stack.

  `stask.push(stack.tos())`

- `store inm`

  Guarda el tope del stack en la variable *inm*.

  `var[inm] = stack.pop()`

- `load inm`

  Carga la variable $inm \in 0.,255$ en el stack.

  `stack.push(var[inm])`

A.3 Parser

```
module Parser where

import UU.Parsing
import UU.Scanner
import Lexer
import AttributeGrammar

type TokenParser a = Parser Token a

-- Parser with starting nonterminal Root
-- Semantic functions generated by UUAG
```

```
pRoot :: TokenParser Root
pRoot
  = Root_Root <$> pDecls <*> pDodecls

pDecls :: TokenParser Decls
pDecls
  = pList pDecl

pDecl :: TokenParser Decl
pDecl
  = (\x xs _ y -> Decl_Function x xs y) <$>
      pVarid <*> pArgs <*> pKey "=" <*> pExpr
    <|> (\x _ y -> Decl_Const x y) <$>
      pConid <*> pKey "=" <*> pExpr

pArgs :: TokenParser [String]
pArgs
  = pList pVarid

pExpr :: TokenParser Expr
pExpr
  = pAdd
  <|> pIfExpr

pIfExpr :: TokenParser Expr
pIfExpr
  = (\_ cond _ t _ e -> Expr_If cond t e) <$>
      pKey "if" <*> pExpr <*> pKey "then" <*> pExpr <*> pKey "else" <*> pExpr

-- Lowest precedence operators.
pBinOp :: TokenParser String
pBinOp
  = pKey "+"
  <|> pKey "-"
  <|> pKey "<"
  <|> pKey ">"
  <|> pKey "<="
  <|> pKey ">="
  <|> pKey "=="
  <|> pKey "/="
  <|> pKey "and"
```

```
  <|> pKey "or"

-- Highest precedence operators
pBinOpH :: TokenParser String
pBinOpH
  = pKey "*"
  <|> pKey "/"

-- Lowest precedence operators expressions.
pAdd :: TokenParser Expr
pAdd
  = pFactor
  <|> (\x op y -> Expr_BinExpr op x y) <$> pFactor <*> pBinOp <*> pExpr

-- Highest precedence operators expressions.
pFactor :: TokenParser Expr
pFactor
  = pTerm
  <|> (\x op y -> Expr_BinExpr op x y) <$> pTerm <*> pBinOpH <*> pFactor

-- (Atom) Simple terms: Numbers, Variables, Constants or parenthized expr.
pTerm :: TokenParser Expr
pTerm
  = (\x -> Expr_Var x) <$> pVarid
    <|> (\x -> Expr_Const x) <$> pConid
    <|> (\x -> Expr_Int $ read x) <$> pInteger16
    <|> (\_ x _ -> x) <$> pKey "(" <*> pExpr <*> pKey ")"

-- do declarations
--
pDodecls :: TokenParser Dodecls
pDodecls
  = (\_ _ x _ -> x) <$>
      pKey "do" <*> pKey "{" <*> pList pDodecl <*> pKey "}"

pDodecl :: TokenParser Dodecl
pDodecl
  = (\_ x y -> Dodecl_Output x y) <$>
      pKey "output" <*> pExpr <*> pVarid
  <|> (\x _ _ y -> Dodecl_Read x y) <$>
      pVarid <*> pKey "<-" <*> pKey "read" <*> pExpr
```

```
  <|> (\x _ _ f s -> Dodecl_Lift x f s) <$>
      pVarid <*> pKey "<-" <*> pKey "lift" <*> pVarid <*> pVarid
  <|> (\x _ _ f s1 s2 -> Dodecl_Lift2 x f s1 s2) <$>
      pVarid <*> pKey "<-" <*> pKey "lift2" <*> pVarid <*> pVarid <*> pVarid
  <|> (\x _ _ f v s -> Dodecl_Folds x f v s) <$>
      pVarid <*> pKey "<-" <*> pKey "folds" <*> pVarid <*> pExpr <*> pVarid
```