UNIVERSIDAD AUTONOMA DE MADRID

ESCUELA POLITECNICA SUPERIOR

Grado en Ingeniería Informática y Matemáticas

TRABAJO FIN DE GRADO

Compilador para lenguaje basado en continuaciones

Carlos Ramos Carreño
Tutor: Jesús Sánchez Cuadrado
Ponente: Juan de Lara Jaramillo

Julio 2015

Resumen

Las continuaciones son una construcción de algunos lenguajes de programación a partir de las cuales pueden elaborarse elementos más complejos, como fibras, generadores, excepciones y corrutinas. Sin embargo, hay poco soporte nativo para ellas en los lenguajes más populares, como por ejemplo Java.

En este trabajo se estudia este concepto en profundidad, centrándonos en las llamadas "continuaciones delimitadas". Para ello se ha implementado un lenguaje de bajo nivel con soporte para continuaciones, que se ejecuta sobre la Máquina Virtual de Java gracias a una compilación a notación CPS y que puede interoperar con programas Java estándar. Asimismo, se ha experimentado realizando ejemplos de construcciones no triviales que se pueden programar en este lenguaje.

Palabras clave: continuacion, JVM, Máquina Virtual de Java, Java, corrutina, excepcion, generador, CPS, Continuation-Passing Style

Abstract

A continuation is a construct of some programming languages that provides seamless support for the implementation of more complex elements, such as fibers, generators, exceptions and coroutines. However, there is little native support for it in popular languages, such as Java.

In this work we study this concept extensively, especially focussing on the so called "delimited continuations". We have implemented a low-level language with support for continuations, which is executed over the Java Virtual Machine thanks to a compiler based on a translation to CPS notation, and thus being able to interoperate with standard Java programs. We have also experimented and tested the language by implementing several examples of non-trivial constructs.

Keywords: continuation, JVM, Java Virtual Machine, Java, coroutine, exception, generator, CPS, Continuation-Passing Style

Agradecimientos

En primer lugar deseo agradecer a mi tutor en este Trabajo de Fin de Grado, Jesús Sánchez Cuadrado, por todo el apoyo que me ha prestado durante la realización del mismo, siguiendo mis progresos y aportando sugerencias muy valiosas. Sin él aún seguiría peleándome con el bytecode y no habría logrado finalizar el proyecto a tiempo.

Quiero dar las gracias también a mis compañeros de carrera. Nuestras numerosas horas de estudio en grupo han sido fundamentales para superar las asignaturas más difíciles, y no sería justo no mencionar su apoyo.

Por último, deseo agradecer a todos los familiares, amigos y profesores que me han prestado su apoyo incondicional a lo largo de estos cinco años de estudio, y en particular agradecer a mis padres que me impulsaran a estudiar aquello que me gusta, y me proporcionaran los medios para ello.

Índice general

Índice de figuras

Glosario

Bytecode	Código de instrucción de la JVM. Ocupa un byte.
Clausura	Función anidada que captura el valor de las variables de la función dentro de la que se define.
Continuación	El resto de la ejecución hasta finalizar el programa. También a veces se usa como sinónimo de continuación reificada.
Continuación delimitada	El resto de la ejecución dentro de un ámbito concreto.
Continuación reificada	Continuación convertida en un objeto de primera clase, que puede ser invocada.
Continuation-Passing Style	Notación en la que las funciones no retornan, sino que se escribe cada llamada a función pasando como parámetro el código que debe ejecutar al acabar (su continuación).
CPS	**C**ontinuation-**P**assing **S**tyle
Corrutina	Generalización de una subrutina en la que se permiten retornos intermedios y continuar ejecutando la función a partir del último retorno intermedio realizado.
Función anónima	Función que no posee nombre, sino que se escribe copiando su definición en el momento en que se usa.
Generador	Construcción de un lenguaje similar a una función que genera el siguiente elemento de una secuencia cada vez que es llamada.
Goto no local	Instrucción de salto no limitada al ámbito de una función, sino que puede saltar de una a otra.
JVM	**J**ava **V**irtual **M**achine. Una máquina virtual de Java.
Lambda	Función anónima.

Capítulo 1

Introducción

La mayoría de lenguajes de programación comparten una serie de elementos básicos, como saltos condicionales o bucles que se pueden usar para construir cualquier tipo de programa. Sin embargo existen construcciones como las excepciones, las fibras o los generadores que no están disponibles en todos los lenguajes, y cuya implementación en lenguajes que no las soporten de forma nativa es compleja.

Una alternativa a implementar cada una de estas características de manera separada es hacer uso del concepto de continuaciones delimitadas, una construcción más general que permite tratar al resto de la ejecución hasta un punto del programa como un objeto de primera clase, pudiendo guardar el contexto de ejecución de un programa, descartarlo y restaurarlo posteriormente. Este tipo de construcción es muy flexible, y permite emular las construcciones indicadas anteriormente (y algunas más) de forma sencilla a partir de ella.

Sin embargo, los lenguajes de propósito general más utilizados, como Java, aún no cuentan con soporte para continuaciones. Con respecto a este último se han intentado algunas aproximaciones para añadir continuaciones, pero la mayoría requieren usar una máquina virtual modificada, con bytecodes específicos.

Además, hoy en día se construyen muchos lenguajes de dominio específico (DSLs) que podrían beneficiarse de las continuaciones usándolas como bloque fundamental sobre el que construir operaciones más avanzadas.

El objetivo de este proyecto será construir un lenguaje de bajo nivel, compatible con Java, para permitir dos cosas: experimentar y estudiar como implementar las continuaciones delimitadas y permitir la creación de lenguajes de más alto nivel que compilen a este lenguaje para tener soporte para continuaciones.

En este proyecto se estudiará el concepto de continuación y se explicarán las posibilidades de programación que ofrece el tener un control directo de las continuaciones en un programa. Asimismo se presentará la notación CPS (Continuation passing style), una forma de escribir código muy usada a la hora de razonar y trabajar con continuaciones.

Se mostrarán intentos anteriores de implementación de continuaciones como objetos de primera clase sobre la máquina virtual de Java y se procederá a realizar una implementación de continuaciones delimitadas sobre una máquina virtual de Java sin modificaciones. Por último se mostrarán los resultados obtenidos mediante la programación de algunas de las construcciones no triviales mencionadas anteriormente, a partir del lenguaje elaborado.

1.1. Estructura del documento

Este documento tiene cuatro secciones principales. En el capítulo 2 se introducen los términos y conceptos que se usan a lo largo del documento. Se presentan las continuaciones delimitadas y la notación CPS, y se exponen posibles usos de las continuaciones delimitadas en programación. También se introducen brevemente las herramientas a usar para la realización del proyecto, principalmente el plugin Xtext y el framework EMF.

En el capítulo 3 se presenta el estado del arte en la implementación de soporte para continuaciones delimitadas sobre la máquina virtual de Java. Se describen las principales aproximaciones (estáticas y dinámicas) para lograr esto y se aportan referencias a los principales trabajos en este ámbito.

El capítulo 4 es la parte central de este trabajo. En él se expone la metodología seguida y se explica el lenguaje que se ha implementado, razonando las decisiones que han llevado a él. Por último se discuten las formas en las que se compila cada elemento del lenguaje con el fin de realizar una implementación correcta de las continuaciones delimitadas.

En el capítulo 5 se ponen ejemplos y pruebas realizados con el lenguaje implementado para comprobar que las continuaciones funcionan correctamente y se pueden crear construcciones no triviales con ellas.

Se recomienda realizar una lectura en orden de todos los capítulos. Es especialmente importante entender correctamente los conceptos presentados en el capítulo 2 para poder comprender los siguientes capítulos.

El capítulo 3 ofrece una pequeña discusión de la forma de implementar continuaciones sobre la JVM pero puede ser saltado en una primera lectura, ya que no es necesario para entender la implementación del lenguaje.

En el capítulo 4 puede obviarse la descripción del lenguaje en una lectura rápida si el lector ha entendido bien los conceptos de `shift` y `reset` y se ve capaz de leer el código sin explicaciones. Es importante, sin embargo, entender el algoritmo de conversión a CPS y la forma de compilar el código, para lo cual puede ser útil apoyarse en los ejemplos del capítulo 5.

Capítulo 2

Contexto y motivación

En esta sección se presentan conceptos que es necesario comprender antes de poder entender el resto del documento. Se define lo que son las continuaciones y se explican brevemente las herramientas de desarrollo usadas.

Por último, una vez entendidos estos conceptos se indican casos de uso de las continuaciones que justifican y motivan el desarrollo de este proyecto.

2.1. Lambdas en Java 8

Para entender los ejemplos y la generación de código, es necesario entender cómo funcionan las funciones anónimas (también llamadas lambdas[1]) en Java 8.

Una función lambda es una función a la que no se le asigna un nombre. En lugar de eso se escribe directamente el cuerpo de la función en el momento en el que se emplea.

Las funciones anónimas fueron añadidas a la versión 8 de Java con el fin de facilitar la definición de funciones pequeñas y simples que puedan ser pasadas a métodos que necesiten algún mecanismo de callback, como por ejemplo para indicar una función de

[1] Esta nomenclatura tiene su origen en el cálculo lambda desarrollado por Alonzo Church para definir lo que significa computar. Church creó un sistema formal parecido a un lenguaje de programación en el que todas las funciones eran anónimas. Para mayor información sobre el cálculo lambda puede consultarse [1].

comparación a una rutina de ordenación o para establecer una función a ejecutar al pulsar un botón. En versiones anteriores de Java, esto solo era posible mediante el uso de clases anónimas[2].

Además de permitir crear una función sin necesidad de darle un nombre, las funciones lambda pueden usar las variables que estén definidas en la función en la que se define la lambda. Esto se conoce como "capturar" el valor de una variable, y a las funciones que realizan esto se las suele llamar clausuras (closure, en inglés). Por tanto las lambdas en Java 8 son clausuras.

Sin embargo, la captura de variables tiene una limitación, y es que captura el valor de la variable, y no la variable en sí, por lo que no se puede modificar el valor de una variable dentro de una función lambda (y el compilador producirá un error si se intenta). Esto puede solucionarse creando una referencia a una variable y modificando el valor de la variable a través de esa referencia.

Por otra parte, gracias a la recolección de basura de la JVM podemos usar y almacenar la lambda tantas veces como se desee, y solo será liberada cuando no se necesite más, sin que debamos realizar ninguna acción adicional.

La sintaxis de las lambdas en Java 8 se muestra en la figura 2.1. En esta figura puede verse una función que acepta un parámetro de tipo `Integer` y retorna ese parámetro incrementado en uno y convertido a `Double`. El compilador es capaz de inferir los tipos de parámetros y retorno en este caso, porque se guarda la lambda en un objeto de tipo `Function<Integer, Double>` que representa cualquier función que acepte un `Integer` y devuelva un `Double`. En caso de que el compilador no fuera capaz de inferir los tipos, éstos se deben indicar explícitamente.

```
Function<Integer, Double> f = (parametro) -> {
      return parametro + 1.0;
};
```

FIGURA 2.1: Lambda en Java 8.

[2] Aunque las funciones lambda podrían haberse definido en función de las clases anónimas se decidió definirlas usando la nueva instrucción bytecode "invokedynamic" para poder variar y mejorar la implementación en el futuro. Para más información consultar [2].

Alternativamente al tipo `Function<tipo-parámetro, tipo-retorno>` utilizaremos también los tipos `Consumer<tipo-parámetro>` y `Supplier<tipo-retorno>` para funciones sin retorno o sin argumentos, respectivamente.

Para más información sobre lambdas en Java 8 puede consultarse [3].

2.2. Continuaciones

Una continuación es una representación abstracta del estado de un programa en un momento dado. Una continuación representa el resto de la computación, es decir el código que continuará ejecutándose después de una determinada instrucción. Por ejemplo, en el ejemplo de la figura 2.2 en Java se muestra en amarillo la instrucción que está siendo ejecutada, y en azul la *continuación actual*.

```
public static void main(String[] args) {
      int a = 0;
      a += 7;
      System.out.println("Hello");
      a++;
      int b = a + 5;
      System.out.println("Bye");
      return;
}
```

FIGURA 2.2: Representación de la continuación actual (en azul) en un momento de la ejecución del código.

2.2.1. Continuaciones delimitadas

Es posible que solo interese considerar el resto de la computación dentro de cierto ámbito. Éste caso es más general que el anterior (que se puede considerar como un caso particular de éste, en el que el ámbito es todo el programa). Entonces decimos que el código que continuará ejecutándose dentro del ámbito es la continuación delimitada por ese ámbito. En la figura 2.3 se muestra un ejemplo, donde los ámbitos son bloques de código marcados con la palabra reset.

```
public static void main(String[] args) {
{
      int a = 0;
      reset {
            a += 7;
            reset {
                  System.out.println("Hello");
                  a++;
            }
            int b = a + 5;
      }
      System.out.println("Bye");
      return;
}
```

FIGURA 2.3: Representación de la continuación delimitada actual para el reset más interno (en azul) en un momento de la ejecución del código.

2.2.2. Reificar continuaciones

Reificar una continuación (o una continuación delimitada) consiste en crear un objeto que represente la continuación y que disponga de un método para invocar el código de la continuación.

Normalmente una continuación reificada de este modo puede ser invocada tantas veces como se desee, y en el momento que se desee (incluso después de terminar la función que lo contiene). Sin embargo, existen casos particulares de continuaciones que imponen restricciones adicionales, como pueden ser las continuaciones de un solo uso ("one shot continuation" en inglés), que solo pueden ser invocadas una única vez. Otro ejemplo son continuaciones que no guardan el estado de la pila y por tanto no pueden ser llamadas una vez que la función en la que se realiza la captura retorna.

Al invocar una continuación reificada se recupera el contexto de ejecución de la continuación que contiene (conceptualmente) la pila de llamadas, las variables locales y el puntero de instrucción. En cambio los objetos globales y los del *heap* no son restaurados.

A continuación se mostrarán instrucciones de programación que implementan distintas formas de reificar continuaciones.

2.2.2.1. Continuaciones de escape

Una continuación de escape es un caso particular de continuación reificada en el que la continuación solo puede ser invocada para escapar de un ámbito a otro más externo. Esto

es así porque estas continuaciones no guardan el estado de la pila, así que no pueden invocarse una vez que la función en la que se realiza la captura finaliza. Como resultado de no capturar la pila, las variables locales que se hayan modificado en memoria después de la captura, conservan su valor modificado.

En muchos lenguajes de programación existen instrucciones que permiten cambiar el contexto de ejecución por uno más externo. Hay que observar que este no es el caso de goto, break o continue, ni el de los if o los bucles, ya que este tipo de instrucciones tienen su ámbito restringido a una función, y no pueden escapar de ese contexto.

Goto no local En algunos lenguajes, como C y sus derivados (C++ o D) existe una función[3] que permite guardar el estado de ejecución de un programa (`setjmp`) y una función que permite reemplazar el estado de ejecución actual por un estado guardado previamente (`longjmp`). Estas funciones implementan, efectivamente, la captura y la llamada de la continuación.

La función `setjmp` en el lenguaje C recibe como parámetro un buffer de tipo `jmp_buf` y lo inicializa con el contexto actual (típicamente incluye la información de los registros). Esta función devuelve 0 al ser llamada de forma normal.

La función `longjmp`, por otra parte, recibe un buffer de tipo `jmp_buf` ya inicializado con un contexto y un valor. La ejecución continúa después de la llamada a `setjmp` que inicializó el bucle, y haciendo que esa función retorne el valor pasado a `longjmp`.

Como ya se ha dicho antes, este tipo de estructura de control tiene la desventaja de que el contexto se pierde cuando se sale del ámbito de la función que llamó a `setjmp`, por lo que no puede usarse para moverse alternando entre varios puntos del código, sino únicamente hacia arriba en la pila de llamadas.

Otra desventaja de esta instrucción es que si una variable local no se guarda en la pila (por ejemplo, porque el compilador decide almacenarla temporalmente en un registro) y es modificada después de guardar el estado, su valor al retornar mediante `longjmp` es

[3] Esta función tiene más limitaciones que una función normal, pues solo se puede usar en expresiones sencillas que no requieran almacenamiento temporal[4].

indeterminado. Para evitar que el compilador realice este tipo de optimización, se puede marcar la variable como `volatile`.

En la figura 2.4 se muestra un ejemplo de uso de estas instrucciones, para retornar de una función que no devuelve nada en caso de error.

```c
#include <setjmp.h>
#include <stdio.h>

jmp_buf continuacion;

void hazAlgo(int valor){

	if(valor == 0) {
		/* Hacer algo */
	}
	else {
		printf("Detectado error\n");
		longjmp(continuacion, 1);
	}
}

int main() {
	volatile int valor = 0;

	if(setjmp(continuacion) == 0) {

		printf("Continuación capturada\n");

		valor++;
		hazAlgo(valor);

		return 0;

	} else {

		printf("Continuación invocada\n");
		printf("El valor es %d\n", valor);

		return 1;

	}
}
```

FIGURA 2.4: Ejemplo de uso de goto no local en C

Después de ejecutar este programa, la salida obtenida es la siguiente:

```
Continuación capturada

Detectado error

Continuación invocada

El valor es 1
```

Excepciones con `try-catch` y `throw` El mecanismo de excepciones usado en lenguajes como C++ o Java es otro ejemplo de escape de una continuación. Cuando un programa ejecuta la instrucción `throw` y esta es capturada se descarta la continuación delimitada por el ámbito entre el `try` y el `catch` que capturan la excepción.

Las excepciones son un ejemplo de lo que se ha mencionado antes de continuaciones "one shot" o de un solo uso. Al no devolver explícitamente una continuación reificada, éstas solo pueden invocarse una sola vez.

Tanto el goto no local como el `try-catch` son funcionalmente equivalentes, y de hecho algunas librerías, como libpng[4], usan `setjmp` y `longjmp` para gestionar los errores.

En la figura 2.5 se muestra el ejemplo de goto no local anterior, realizado usando excepciones en Java.

La salida es la misma que en el caso anterior.

2.2.2.2. Call-with-current-continuation (`call/cc`)

La instrucción `call/cc` tiene su origen en el lenguaje Scheme. Esta instrucción recibe una función y la llama pasándole como parámetro un objeto que representa la continuación reificada. Este objeto puede ser invocado como una función pasándole un parámetro, y entonce la ejecución del programa retornará de la función `call/cc` devolviendo ese valor y continuará ejecutando el programa. La continuación reificada puede invocarse tantas veces como se desee.

En la figura 2.6 se muestra un ejemplo de la instrucción `call/cc` en el lenguaje Ruby. En este ejemplo se imprime "Hola mundo" seguido de las frases "Tu turno" y "Mi turno" alternándose en un bucle infinito.

[4]libpng es la librería libre de referencia para leer y escribir imágenes en formato PNG.

```
private static void hazAlgo(int valor) throws Exception {

      if (valor == 0) {
            /* Hacer algo */
      } else {
            System.out.println("Detectado error");
            throw new Exception();
      }
}

public static void main(String[] args) {

      int valor = 0;

      try {
            System.out.println("Continuación capturada");

            valor++;
            hazAlgo(valor);

            System.exit(0);

      } catch (Exception e) {

            System.out.println("Continuación invocada");
            System.out.println("El valor es " + valor);

            System.exit(1);

      }
}
```

FIGURA 2.5: Ejemplo de uso de excepciones en Java

```
require "continuation"

def a
  puts "Hola mundo"
  callcc {|cc| $label1 = cc } # Similar a "label:"
  puts "Tu turno"
  b
end

def b
  puts "Mi turno"
  $label1.call # Similar a "goto label"
end

a
```

FIGURA 2.6: Ejemplo de uso de call/cc en Ruby

2.2.2.3. Reificar continuaciones delimitadas con `shift/reset`

Una forma de reificar continuaciones delimitadas es usar un par de instrucciones relacionadas: la instrucción `reset` y la instrucción `shift`.

La instrucción `reset` recibe una función o bloque de código y ejecuta esa función como un contexto para las continuaciones delimitadas.

La instrucción `shift` recibe una función que acepta como parámetro la continuación delimitada actual. Ejecuta esa función y escapa de la continuación actual al `reset`.

En la figura 2.7 se muestra un ejemplo en Scala que hace uso de las instrucciones `shift` y `reset`. En este ejemplo la función `foo` devolvería en principio el resultado de evaluar el `shift` más uno. En cambio, al evaluar el shift se invoca la función pasada a éste empleando como argumento la continuación delimitada actual. En este caso la continuación delimitada actual recibiría como parámetro el valor retornado por la expresión actual (al ser un lenguaje que permite expresiones), lo incrementaría en uno y al resultado de ésto lo multiplicaría por dos (ya que esas son las instrucciones a ejecutar antes de salir del reset).

```
import scala.util.continuations._

object pruebas extends App {
  def foo(): Int @cps[Int] = {
    shift { k: (Int=>Int) =>
      k(7)
    } + 1
  }

  println(reset(2 * foo())) // Imprime 16
}
```

FIGURA 2.7: Ejemplo de uso de shift/reset en Scala

Ahora bien, dentro de la función pasada al shift lo que se hace es retornar el valor de la continuación recibida evaluada en 7. Por tanto, se incrementa 7 en uno obteniendo 8 y se multiplica por dos obteniendo 16. Como el shift retorna este valor y descarta la continuación, éste será el valor retornado por reset, que es el que se imprime.

2.2.3. Continuation-Passing Style (CPS)

La notación CPS (abreviatura de continuation-passing style) es una forma de escribir el código de un programa en la que las continuaciones se escriben y se pasan de forma explícita.

Cuando se escribe un programa en notación CPS, cada función recibe un parámetro adicional, que representa la continuación de la función. En lugar de retornar, la función invocará la continuación recibida pasando el valor de retorno. De esta forma, las funciones nunca regresan al código que las llamó, sino que la ejecución del programa transcurre "hacia adelante" sin retornar hasta que el programa finalice.

Cuando desde una función A se desea invocar a una función B, se pasa como continuación de la función B el código que se desea ejecutar a continuación (incluyendo la invocación de la continuación de la función A).

En la figura 2.8 se muestra un ejemplo en Java de notación CPS. En este caso tenemos una función A que acepta un parámetro de tipo `Integer` y "devuelve" (llamando a la continuación con él) un parámetro de tipo `String`. Dentro del cuerpo de la función se incrementa el parámetro en uno y se convierte en `Double` pasándoselo a una función B que acepta un parámetro de tipo `Double` y "devuelve" un objeto que soporta el método `toString`.

```
public void A(Consumer<String> continuacion, Integer parametro) {
      Double valor = parametro + 1.0;

      B((p) -> {
            continuacion.accept(p.toString());
      }, valor);
}
```

FIGURA 2.8: Ejemplo de notación CPS en Java

Es importante notar aquí dos cosas. La primera es que se pasa la invocación a la continuación de la función A como parte de la continuación de la función B, con el fin de ejecutarla después. La segunda es que, para que esto fuera una notación CPS real las funciones Java, como `toString` o la suma, también deberían estar en notación CPS. Como esto es imposible, nosotros las trataremos como si fueran primitivas o instrucciones básicas.

El lector atento puede haber notado que tal y como se ha descrito la notación CPS ocasiona que el tamaño de la pila crezca con cada llamada a función, con el peligro de desbordamiento que eso conlleva. Por ello, cualquier implementación que pretenda escribir el código en notación CPS para ser ejecutado emplea la optimización conocida como tail-call optimization (TCO) que en español se traduce como optimización de cola. Esta optimización consiste en aprovechar que el marco de pila[5] de la función actual al realizar una llamada a función no va a volver a ser usado (ya que las funciones no retornan) y por tanto puede ser reutilizado como marco de pila para la función invocada, evitando por tanto el crecimiento desmedido de la pila.

2.3. Herramientas para la creación de lenguajes

Para crear el parser de nuestro lenguaje, hemos decidido usar la herramienta Xtext, que forma parte del paquete de desarrollo de software dirigido por modelos de Eclipse.

Xtext es una herramienta que a partir de un modelo de la sintaxis abstracta de un lenguaje (que define los conceptos básicos del lenguaje y los elementos que los forman) y de una descripción de la sintaxis concreta (de forma muy similar a Yacc/Bison) nos permite crear un parser del lenguaje y un plugin para Eclipse que incluye un editor (con autocompletado y resaltado de sintaxis) de nuestro lenguaje. Para ello Xtext se apoya en dos tecnologías previas: el framework EMF (Eclipse Modelling Framework) para definir modelos Ecore que definen la sintaxis abstracta y el generador de parsers ANTLR que se usa para crear el parser de nuestro lenguaje.

Para definir el modelo de la sintaxis abstracta, usaremos el formato de fichero Emfatic, que permite hacerlo mediante un fichero de texto, sin necesidad de hacerlo a través de menús.

[5] El marco de pila (en inglés stack frame) de una función es la porción de la pila que contiene toda la información local de esa función: argumentos, dirección de retorno y variables locales.

2.4. Aplicaciones

Las continuaciones permiten crear múltiples construcciones en el lenguaje que son complicadas de realizar de otra forma. Normalmente son los programadores de bibliotecas los que crean estas construcciones y las ofrecen al usuario, ya que programar con continuaciones es complicado y es fácil cometer errores.

2.4.1. Excepciones

Como ya se comentó en la sección 2.2.2.1 las excepciones y el goto no local pueden verse como casos particulares de continuaciones reificadas. Si un lenguaje soporta continuaciones, o continuaciones delimitadas es posible implementar excepciones basadas en ellas. También es posible implementar excepciones aprovechándose de que la instrucción `shift` descarta la ejecución hasta el final del `reset`. En la sección 5.1 se mostrarán ejemplos de ambos casos usando el lenguaje elaborado en este proyecto.

2.4.2. Generadores

Los generadores son construcciones muy parecidas a los iteradores, en el sentido de que se les puede pedir elementos de forma secuencial. Sin embargo, los iteradores suelen recorrer una colección finita, ya existente, de elementos, mientras que los generadores recorren una sucesión potencialmente infinita, y solo generan el elemento correspondiente cuando es necesario (evaluación perezosa). Esto hace que los generadores sean más eficientes para ciertas tareas que requieran recorrer un número muy grande de elementos en sucesión.

En la figura 2.9 se muestra un ejemplo de generador de la serie de los números de fibonacci, implementado en Python.

```
def fibonacci(limit):
    a, b, c = 0, 1, 0
    while c < limit:
        yield a
        a, b, c = b, a+b, c+1

for number in fibonacci(100):
    print(number)
```

FIGURA 2.9: Ejemplo de uso de un generador en Python

2.4.3. Corrutinas

Las corrutinas son una generalización de las funciones, en las que se permite retornar en varios puntos (mediante una instrucción usualmente llamada `yield`) y reanudar la ejecución en la siguiente llamada a partir del punto en el que se retornó. Las corrutinas pueden implementarse mediante continuaciones delimitadas, retornando mediante un `shift` una continuación que podrá invocarse posteriormente y continuará la ejecución desde donde se lanzó.

Las corrutinas, a su vez, sirven para implementar tanto los casos mostrados anteriormente como otras funcionalidades avanzadas, que mostramos a continuación.

2.4.3.1. Fibras

Las fibras son un mecanismo de concurrencia similar a los hilos (threads) pero con la diferencia de que son manejados a nivel de usuario, y pasar de ejecutar una a ejecutar otra se hace de forma no expulsiva (non preemptive) sino que cada fibra debe ceder la ejecución a otra explícitamente mediante una instrucción `yield`, es decir de forma cooperativa.

Esto hace que sea un mal mecanismo de concurrencia si no se puede confiar en el código que ejecutan las otras fibras, pues podrían no ejecutar una instrucción `yield` y por tanto impedir la ejecución de las demás fibras[6].

Sin embargo, dentro de un mismo proceso es típico que las fibras sean de confianza, y el uso de fibras y entrada/salida no bloqueante puede mejorar el rendimiento cuando hay muchas fibras involucradas, al no tener que cambiar a modo kernel para realizar un cambio de contexto ni necesitar usar primitivas de sincronización (ya que solo se puede ejecutar una fibra a la vez y éstas dejan la ejecución cuando lo deseen)[7].

[6] A pesar de ello han existido sistemas operativos que presentaban multihilo cooperativo con mayor o menor éxito, como Windows 3.1.

[7] Muchos sistemas operativos proporcionan algún tipo de soporte para trabajar con fibras. Por ejemplo Windows tiene la función `CreateFiber` y los sistemas Unix suelen disponer del header `ucontext.h`, con funciones para administrar contextos de ejecución.

2.4.3.2. Máquinas de estados

Cuando se desean programar máquinas de estados, una posible alternativa al uso del `goto` es emplear corrutinas que representan cada estado y van pasando el control de una a otra en función de la entrada.

2.4.3.3. Ejecutar una función en varios ciclos de actualización

Algunas aplicaciones, como los videojuegos, presentan un bucle principal que ejecuta para cada objeto una función de actualización cada cierto tiempo (por ejemplo, cada vez que se repinta la pantalla o se hacen los cálculos de la física del videojuego). Es muy frecuente en este caso querer ejecutar funciones que comienzan en unos de estos ciclos de actualización y cuya ejecución se prolonga varios ciclos.

Si esta función se ejecutara directamente, impediría ejecutar las actualizaciones de los demás objetos del videojuego durante todo este tiempo, lo que claramente se desea evitar. En lugar de esto puede convertirse esta función en una corrutina, ejecutando un `yield` cada poco tiempo (o instrucciones más avanzadas si el lenguaje lo permite) y permitiendo que el ciclo de actualización se realice para todos los objetos. Cuando la corrutina sea invocada en los siguientes ciclos, continuará su ejecución en el punto en que se suspendió, extendiéndose de esta forma durante varios ciclos.

En la figura 2.10 se muestra un ejemplo de corrutina en Unity 3d, un motor de videojuegos. Esta corrutina permite reducir gradualmente la opacidad de un objeto hasta volverse invisible, a lo largo de varios ciclos de actualización.

```
function Fade() {
    for (var f = 1.0; f >= 0; f -= 0.1) {
        var c = renderer.material.color;
        c.a = f;
        renderer.material.color = c;
        yield WaitForSeconds(0.1);
    }
}
```

FIGURA 2.10: Ejemplo de corrutina en el motor de videojuegos Unity 3d

Si en lugar de realizar la instrucción `yield` se realizara la espera dentro de la función, el resto del juego quedaría esencialmente parado hasta que esta función termine. Por otra

parte, si no se realizara la espera el objeto pasaría de ser completamente opaco a ser invisible en un solo ciclo de actualización.

2.5. Objetivo

El objetivo de este proyecto es intentar crear un lenguaje de programación que se ejecute sobre la máquina virtual de Java y que implemente las instrucciones de captura y ámbito de continuaciones delimitadas (`shift`/`reset`) y la creación de programas de prueba que muestren el funcionamiento del lenguaje implementando algunas de las construcciones que son posibles gracias a las continuaciones.

Capítulo 3

Estado del arte

En este capítulo se describen algunas de las implementaciones que se han realizado de continuaciones sobre una versión (original o modificada) de la JVM.

En general, existen dos formas de acercarse a este problema: la forma dinámica (copiar el estado en tiempo de ejecución) y la forma estática (compilar el programa de forma especial para reificar las continuaciones).

La forma dinámica requiere copiar el estado de los registros y, sobre todo, requiere copiar la pila del programa. Por otra parte, la forma estática puede implementarse mediante la notación CPS.

3.1. Continuaciones usando la pila

Una posible opción para implementar continuaciones consiste en copiar el estado actual del programa total o parcialmente. Dado que el estado del programa incluye no solo la información de los registros, sino también la información de la pila (argumentos, variables locales y llamadas a funciones previas) esta implementación requiere capacidad para interactuar con la pila y poder copiarla.

Un problema a la hora de realizar esta aproximación sobre la JVM es que no existen bytecodes para manipular la pila libremente, sino que se limitan a insertar y extraer

valores de la parte superior de la pila. Esto hace que emplear esta técnica para implementar continuaciones solo sea posible para lenguajes que no funcionen usando la JVM o para aquellos que construyan su propia versión de la JVM.

3.1.1. Aproximación ingenua

La forma ingenua de reificar una continuación es copiando la pila entera y los registros en el momento de la captura de la continuación (en la instrucción shift o call/cc, dependiendo de si las continuaciones son delimitadas o no). De este modo ejecutar una continuación puede realizarse restaurando los registros y modificando el puntero de pila para que apunte a la pila guardada.

Sin embargo, esta aproximación tiene varias desventajas. La primera, es que se puede copiar contexto que no es necesario. Por ejemplo, si una continuación se ejecuta en el mismo marco de pila en el que se captura, solo habría sido necesario copiar la información de ese marco de pila, que es el único que puede ser modificado entre ambos puntos del código. Sin embargo se ha copiado toda la pila, con un gasto innecesario en tiempo y memoria.

Por otra parte, si se reifican varias continuaciones que comparten el estado de parte de la pila, la parte compartida se copiará varias veces, pese a que nunca va a ser modificada.

En el artículo de Iulian Dragos et al.[5] se realiza una implementación de una JVM con soporte para continuaciones, usando esta técnica. Es interesante resaltar que en el artículo se discute la interacción de las continuaciones con otras construcciones del lenguaje Java y la JVM, como excepciones, monitores y modelo de seguridad de la JVM.

3.1.2. Copia bajo demanda (Continuaciones perezosas)

Una forma más eficiente de reificar una continuación es la descrita en el artículo de Lukas Stadler et al.[6]. También es posible consultar la presentación que realizó sobre el tema en la JVM Language Summit[7].

En este artículo se describe una forma de implementar continuaciones que mitiga los problemas de la aproximación anterior. Lo que se propone es copiar cada marco de pila solo cuando sea necesario.

Para ello en la captura de una continuación, se guarda solo el marco de pila actual, pero se intercepta la dirección de retorno haciendo que antes de retornar de cada marco de pila se copie el nuevo marco y se haga que las continuaciones apunten a la copia. De esta forma, existirá como mucho una copia para cada marco de pila, y solo en caso de que el original sea modificado. Las continuaciones, por tanto, se van agrupando en una estructura de árbol, unidas por los marcos de pila que tienen en común.

3.2. CPS (total o parcial)

Como se dijo anteriormente, en la notación CPS las continuaciones se pasan de forma explícita. Por tanto, la continuación ya está reificada, ya que se recibe de la función anterior como una función anónima.

Por tanto, si se escribe el programa en forma CPS implementar call/cc se puede hacer mediante una simple función de biblioteca, que reciba además de la continuación una función a invocar con ella.

Por otra parte, para implementar continuaciones delimitadas mediante shift/reset se puede hacer un planteamiento similar. En lugar de escribir todo el programa en notación CPS, se puede convertir a CPS solamente el bloque de código que esté dentro de un reset, manteniendo el resto del programa con la forma usual. Entonces shift puede implementarse mediante una función de biblioteca que retorne un valor, haciendo que el resto de funciones en CPS retornen el resultado de invocar a su continuación.

Una aproximación parecida para implementar shift/reset utilizando CPS parcial es la empleada por el lenguaje de programación Scala, que también se ejecuta sobre la JVM. Los detalles concretos de esta implementación pueden consultarse en el artículo de Rompf et al.[8].

Capítulo 4

Diseño y desarrollo

En esta sección se explica el proceso que se ha seguido para la elaboración del proyecto, así como las decisiones tomadas en su implementación.

4.1. Metodología y planificación

Este proyecto ha sido realizado entre los meses de Septiembre a Junio del curso académico 2014-2015.

El proyecto se estructuró en varias fases, durante las cuales la comunicación con el tutor se realizó mediante correo electrónico y reuniones periódicas. Se acordó que todo lo desarrollado se añadiría a un repositorio de Git[1], en el servidor Bitbucket, al que ambos tuviéramos acceso. Al haber un único desarrollador, no se realizaron ramas en el repositorio.

[1]Git es un sistema de control de versiones, creado por Linux Torvalds (autor también del kernel del sistema operativo GNU/Linux). A diferencia de otros sistemas de control de versiones, Git permite mantener a los usuarios una copia entera en local y trabajar de forma distribuída, pudiendo posteriormente combinar los cambios locales.

4.1.1. Primera fase: Estudio previo

En esta fase el objetivo era comprender el concepto de continuación, entender el objetivo del proyecto y buscar toda la información necesaria para llevarlo a cabo. Se recopilaron artículos y notas sobre continuaciones y como implementarlas sobre la Máquina virtual de Java, con la ayuda del tutor.

4.1.2. Segunda fase: Tecnologías a usar

El objetivo de esta fase era aprender a usar las herramientas que servirían para desarrollar el proyecto. En esta fase se aprendió el uso básico de Xtext y Emfatic, intentando desarrollar un pequeño lenguaje de prueba de alto nivel inventado, llamado Splang.

En esta fase se pensó desarrollar el proyecto escribiendo directamente código bytecode de Java, mediante el uso de la biblioteca BCEL[2]. Para ello se estudió el funcionamiento interno de la JVM, principalmente mediante la lectura de [9] y [10]. También se comenzó a realizar un compilador de prueba de Splang usando BCEL.

El análisis realizado en esta fase llevó a averiguar que no era posible implementar el copiado de la pila en la JVM, por limitaciones del propio bytecode, así como a comprobar que implementar funciones lambda (para hacer una conversión a CPS) en BCEL era extremadamente complicado.

Por estas razones, se decidió implementar las continuaciones mediante transformación parcial a notación CPS y compilando a código Java (ya que Java 8 soporta funciones lambda). En lugar de crear un lenguaje de alto nivel como Splang, se optó por hacer un lenguaje de bajo nivel, al que llamamos Cont, donde cada instrucción es una línea (a modo de ensamblador). De este modo sería más sencillo realizar la transformación CPS (al no tener que preocuparse por analizar expresiones complejas) y en el caso de que se deseara un lenguaje de alto nivel se podría realizar uno que compile al lenguaje Cont.

[2]BCEL es una biblioteca Java creada y mantenida por Apache que ofrece clases y funciones para manipular clases Java o crear nuevas, permitiendo editar el código bytecode. Usar esta librería permite que el programador no necesite conocer la organización exacta de las clases Java, y le facilita tareas habituales, pero aún así es necesario conocer bien el funcionamiento del bytecode.

4.1.3. Tercera fase: Implementación

En esta fase se realizó la implementación del lenguaje, y se crearon ejemplos de su uso. Aquí fue necesario analizar como implementar correctamente la notación CPS parcial para cada construcción del lenguaje.

La implementación se realizó en Xtext mediante la compilación a código de Java 8. Para comprobar el funcionamiento correcto, se realizaron pruebas de implementación de generadores y corrutinas mediante continuaciones.

4.1.4. Cuarta fase: Memoria

Por último se documentó todo el proyecto mediante la creación de este documento, a partir de toda la información creada con anterioridad. La memoria se ha realizado en Latex a partir de una plantilla para la elaboración de tesis, siendo evaluada periódicamente por el tutor.

4.2. Diseño del lenguaje

El lenguaje desarrollado es un lenguaje de bajo nivel, similar en cierto modo a ensamblador, en el que cada sentencia es una instrucción. Sin embargo, a diferencia de este existen instrucciones que requieren bloques de código, como if o while.

La razón por la que se ha decidido realizar un lenguaje de bajo nivel es para facilitar el parsing y la conversión a CPS. Si se deseara implementar un lenguaje de alto nivel con soporte para continuaciones podría realizarse compilando primero a éste y posteriormente compilando el código resultante usando este compilador.

Este lenguaje, como se ha comentado con anterioridad, es compilado a código Java e implementa continuaciones delimitadas mediante una notación CPS parcial, aprovechando las funciones lambda de Java 8.

Se ha decidido que en este lenguaje el paso sea siempre por referencia, lo que permite de forma sencilla retornar varios valores desde una función, o hacer que una función cambie sus parámetros.

Los tipos básicos de este lenguaje serán `Int`, `String` y `Continuation`. Los tipos `Int`, y `String` son equivalentes a los de cualquier otro lenguaje de programación, mientras que el tipo `Continuation` representa una continuación reificada. Este tipo es, por tanto, esencial en este lenguaje ya que es el tipo del objeto pasado a la instrucción `shift` como continuación actual.

4.2.1. Declaración de programa y clases

Para escribir un programa en el lenguaje Cont, primero se ha de escribir el nombre del programa de la siguiente forma:

```
program <nombre>
```

A continuación, se han de escribir las clases que lo forman, de la siguiente forma:

```
class <nombre> {
    <registros-atributos>

    <métodos>
}
```

La declaración de un registro (ya sea atributo o local a un método) se realiza de la siguiente forma:

```
<nombre> : <tipo>
```

donde tipo puede ser `Int`, `String`, `Continuation` o cualquier tipo de clase Java.

La declaración de un método es la siguiente:

```
def <nombre> [( <parámetro-1>... )] : <tipo>
    <registros-locales>

    <sentencias>
end
```

4.2.2. Instrucciones disponibles

El lenguaje implementado consta con un subconjunto de instrucciones lo suficientemente grande como para poder implementar algoritmos interesantes que usen continuaciones, pero no pretende disponer de todas las instrucciones de lenguajes de más alto nivel, como Java. Muchas instrucciones de estos lenguajes, como `for` o `do-while` no se han implementado al no ser necesarias para alcanzar el objetivo de este proyecto.

Las instrucciones disponibles son las siguientes:

4.2.2.1. Instrucciones de movimiento

`mov` *<reg-destino> <literal-de-cadena>*

`mov` *<reg-destino> <literal-de-entero>*

`mov` *<reg-destino> <reg-origen>*

Estas instrucciones asignan al registro recibido como primer parámetro, el valor recibido como segundo parámetro.

4.2.2.2. Instrucciones de operación

`add` *<reg-destino> <reg-sumando-1> <reg-sumando-2>*

`sub` *<reg-destino> <reg-minuendo> <reg-sustraendo>*

`mul` *<reg-destino> <reg-multiplicando-1> <reg-multiplicando-2>*

`div` *<reg-destino> <reg-dividendo> <reg-divisor>*

`and` *<reg-destino> <reg-operando-1> <reg-operando-2>*

`or` *<reg-destino> <reg-operando-1> <reg-operando-2>*

`not` *<reg-destino> <reg-operando>*

Estas instrucciones guardan en el registro pasado como primer parámetro el resultado de realizar la operación correspondiente con los restantes.

Todas estas instrucciones trabajan con registros de tipo entero (ya que no hay booleanos nativos en este lenguaje).

4.2.2.3. Instrucciones de comparación

`less` *<reg-destino> <reg-1> <reg-2>*

`greater` *<reg-destino> <reg-1> <reg-2>*

`equals` *<reg-destino> <reg-1> <reg-2>*

Estas instrucciones guardan en el registro pasado como primer parámetro el resultado de realizar la comparación correspondiente de los otros dos registros.

Las instrucciones less y greater requieren que los registros implementen la interfaz Comparable de Java (por defecto los enteros y cadenas lo hacen).

4.2.2.4. Instrucción de retorno de función

`return` *<reg-retorno>*

Esta instrucción hace que la función deje de ejecutarse y retorne un valor, de forma similar al funcionamiento de `return` en otros lenguajes de alto nivel.

4.2.2.5. Instrucción de llamada a método

`call` [*<reg-destino>* `<-`] [*<reg-objeto>*`.`]*<nombre-metodo>* [*<reg-parámetro-1>*...]

Esta instrucción invoca al método indicado del objeto especificado. Si no se indica ningún objeto, se usará el método del objeto actual. Es posible invocar un método estático escribiendo como nombre de método la invocación completa entre comillas, por ejemplo "`System.out.println`". En el caso de que se especifique registro destino, se guardará en él el valor de retorno del método.

4.2.2.6. Instrucción de salto condicional

```
if <reg-condición> {
    <sentencias>
}
[ else {
    <sentencias>
} ]
```

Esta instrucción es similar al `if` de cualquier lenguaje de programación de alto nivel, con la salvedad de que por simplificar no admite ramas `elsif`.

Como este lenguaje es de bajo nivel, no hay expresiones, por lo que la condición viene dada por un registro. Si el registro es 0 la condición es falsa y en caso contrario la condición es verdadera.

4.2.2.7. Instrucción de bucle

```
while <reg-condición> {
    <sentencias>
}
```

Este es el único bucle que soporta nuestro lenguaje. Los demás tipos de bucle, como el `for` o el `do-while` pueden implementarse en función de éste.

Al igual que en el `if` la condición viene dada por un registro. Al ser éste un lenguaje en el que se usan referencias en lugar de valores, el registro de condición puede modificarse dentro del bucle.

4.2.2.8. Instrucción de ámbito de continuación delimitada (reset)

```
reset <reg-retorno> {
    <sentencias>
}
```

Esta instrucción corresponde al `reset` de las continuaciones delimitadas. Permite establecer un ámbito, dentro del que se restringen las continuaciones. Si se ejecuta una instrucción `shift` dentro de ese ámbito, siendo éste el reset más interno que la contiene,

el valor devuelto por la instrucción `shift` se guarda en el registro de retorno del reset y la ejecución continúa tras el reset.

4.2.2.9. Instrucción de captura de continuación delimitada (shift)

```
shift <reg-continuación> {
    <sentencias>
}
```

Esta instrucción representa el `shift` de las continuaciones delimitadas. En la mayoría de los lenguajes que la soportan, esta instrucción recibe como parámetro una función lambda que tiene como único parámetro la continuación delimitada actual. Nuestro lenguaje no tiene funciones lambda, así que esta instrucción recibe como parámetro la declaración del registro que contendrá la continuación delimitada actual y un bloque de código a ejecutar (en el que debe haber un `return` de lo que se devolverá al reset).

4.3. Implementación del lenguaje

En esta sección se explica cómo se realiza la compilación del lenguaje a código Java, y especialmente cómo se implementan las continuaciones mediante notación CPS parcial.

Se describirá para cada instrucción cómo se realiza el paso a código Java, tanto si se encuentra en notación CPS como si no.

4.3.1. Biblioteca de tipos y funciones

El código Java generado usa una serie de tipos de datos y funciones comunes que se han decidido agrupar en una clase Java llamada CPS, que se distribuye con el compilador.

Esta clase contiene la implementación de las referencias, de la clase Continuation y de las implementaciones continuables de las instrucciones de `while`, `if`, `reset` y `shift`.

4.3.2. Conversión en el árbol sintáctico de una lista de sentencias a notación CPS

En varias ocasiones en el código es necesario modificar el árbol sintáctico para cambiar un conjunto de sentencias en notación usual a notación CPS. Para realizar ésto, se ha creado una función que recibe una lista de sentencias y la pasa a notación CPS.

El paso a CPS sigue el siguiente algoritmo básico:

- Buscar la primera sentencia que acepte una continuación. Las sentencias que aceptan continuaciones son `if`, `while`, `call`, `reset` y `shift`.
- Si no se encuentra una sentencia de ese tipo, el proceso ha finalizado.
- Obtener todas las sentencias posteriores a la encontrada y añadirlas como continuación de la sentencia encontrada.
- Si la sentencia encontrada es un `if` copiar su continuación al final del cuerpo de sus bloques de código a ejecutar en el caso de que la condición se cumpla y en el caso de que no (incluso aunque el bloque else esté vacío). Pasar a CPS ambos bloques.
- Si la sentencia encontrada es un `while`, pasar a CPS su cuerpo.

No es necesario pasar a CPS el cuerpo de una instrucción `reset`, porque ésto se hace al encontrar esa instrucción al recorrer el árbol.

No se debe pasar a CPS el cuerpo de un shift, ya que esta instrucción debe realizar un retorno normal, y no se pueden anidar.

4.3.3. Registros

Las declaraciones de registros en el lenguaje, se convierten en el código Java generado en declaraciones de objetos de tipo referencia a objeto, usando la clase genérica Reference definida en la clase CPS. Esta clase se usa para encapsular los objetos reales de forma que se pasen por referencia a las funciones y métodos. El código que define esta clase se muestra en la figura 4.1.

```
static public class Reference<T> {
      T value;

      public Reference() {
            this(null);
      }

      public Reference(T value) {
            this.value = value;
      }

      public T getValue() {
            return value;
      }

      public void setValue(T value) {
            this.value = value;
      }

      @Override
      public java.lang.String toString() {
            return "" + value;
      }
}
```

FIGURA 4.1: Clase Reference

Por ejemplo, la sentencia de la figura 4.2 se transformará, a la hora de compilar a código Java en la sentencia de la figura 4.3. Esto es independiente de dónde se declare el registro y de si el código es continuable o no.

```
numero : Int
```

FIGURA 4.2: Ejemplo de declaración de un registro

```
CPS.Reference<Integer> numero = new CPS.Reference<Integer>();
```

FIGURA 4.3: Código Java generado al declarar un registro

En este caso, el tipo `Int` es reconocido como la clase Java `Integer`, siendo éste, junto con el tipo `Continuation` los únicos tipos que no se traducen a una clase Java con su mismo nombre. En el caso del tipo `Continuation`, la clase traducida es `CPS.Continuation`. En el resto de los casos, la referencia es a una clase cuyo nombre concide con el del tipo referenciado.

4.3.4. Clases

Cada clase de nuestro lenguaje se convierte de forma trivial a una clase Java, salvo por los métodos. Por ejemplo, el programa de la figura 4.4, que solo consta de una clase vacía

se traduce en código Java al programa de la figura 4.5.

```
program Programa

class Clase

end
```

FIGURA 4.4: Ejemplo de clase

```
import java.util.function.Function;

public class Clase{
}
```

FIGURA 4.5: Código Java generado al crear una clase

La instrucción `import` generada sirve para poder usar, cuando se realice el paso a CPS, la interfaz funcional `Function`. Esta interfaz representa una función lambda con un único parámetro y un valor de retorno, que será la forma que usaremos para representar una continuación. El valor pasado a la continuación será el valor retornado por la función anterior, y el retorno será el propagado por una instrucción `shift`.

4.3.5. Métodos

La traducción de los métodos a Java no es tan trivial como las explicadas anteriormente. Como no sabemos si los métodos se van a invocar en un fragmento de código en notación CPS o no, realizamos dos implementaciones Java del método. La primera implementación es la normal, con el mismo número de parámetros y valor de retorno que como se declaró en nuestro lenguaje. En cambio, la segunda implementación es en notación CPS. Esta implementación recibe como parámetro adicional (primer parámetro de la función) la continuación actual, y retorna el resultado de invocar dicha continuación.

Para crear la segunda versión del método, se clona el método usando las funciones de copia proporcionadas por el framework EMF, y posteriormente esa copia se pasa a CPS de la forma explicada en la sección 4.3.2.

En la figura 4.6 se observa la declaración de una clase con un método, como se haría en el lenguaje implementado. En la figura 4.7 se puede observar el resultado de compilar esa clase a código Java.

Se puede observar que la versión no-CPS es muy parecida a la declaración original, salvo por el hecho de que la función recibe y devuelve referencias.

```
program Programa

class Clase

      def metodo(numero : Int) : Int
            return numero
      end

end
```

FIGURA 4.6: Ejemplo de clase con un método

```
import java.util.function.Function;

public class Clase{

      public CPS.Reference<Integer> metodo(CPS.Reference<Integer> numero){
            return numero;
      }

      public CPS.Reference<?> metodo(Function<CPS.Reference<Integer>,
            CPS.Reference<?>> CPS_continuacion,
            CPS.Reference<Integer> numero){
            return CPS_continuacion.apply(numero);
      }
}
```

FIGURA 4.7: Código Java generado al crear una clase con un método

En cambio, la versión CPS devuelve una referencia a cualquier objeto. Eso es así porque no sabemos qué objeto se devolverá desde el shift. Además, se acepta como parámetro una función (lambda) que representa a la continuación, y que recibe el retorno de ésta función (por lo que el tipo de su parámetro es el mismo que el tipo de retorno de esta función) y devuelve también una referencia a cualquier objeto, proveniente de un shift.

Aprovechamos este ejemplo para ver cómo se compila la instrucción `return` cuando se encuentra en código en notación CPS y cuando no.

Cuando un return no está en código en notación CPS, se traduce a un `return` normal de Java. En cambio, en código CPS `return` se traduce a un `return` del resultado de invocar la continuación recibida pasando como argumento el valor a retornar por la función. Este valor retornado por la continuación será el valor devuelto por un shift.

Por último, si se crea un método con el nombre `main` el compilador automáticamente generará un método `main` de Java que cree un objeto de este tipo e invoque la función `main` en él.

4.3.6. Instrucciones

A continuación se explicará, para cada tipo de instrucción, como se traduce a Java, tanto cuando el código está con la notación usual como cuando se usa en un bloque CPS.

4.3.6.1. Instrucciones de movimiento

Las instrucciones de movimiento se compilan igual tanto si el código está en notación CPS como si no. El código Java que se genera se limita a hacer un set de la referencia del primer registro al valor del segundo operando.

En la figura 4.8 puede verse como se compilaría una instrucción de movimiento de un registro a otro (en la primera línea) a código Java (en la segunda línea). El resto de instrucciones de movimiento son similares.

```
mov destino origen
```

```
destino.setValue((Integer)origen.getValue());
```

FIGURA 4.8: Compilación de instrucción de movimiento

El casting se realiza para poder trabajar con los objetos devueltos por un shift (de los que se desconoce su tipo).

4.3.6.2. Instrucciones de operaciones con enteros

Las instrucciones de operaciones con enteros también se compilan igual tanto si el código está en notación CPS como si no. El código Java que se genera se limita a hacer un set de la referencia del primer registro al valor resultado de la operación de los registros restantes.

En la figura 4.9 puede verse como se compilaría una instrucción suma de los registros. Para compilar cualquier otra basta con cambiar el operador utilizado.

```
add destino entero1 entero2
```

```
destino.setValue(entero1.getValue() + entero2.getValue());
```

FIGURA 4.9: Compilación de instrucción de suma

4.3.6.3. Instrucciones de operaciones lógicas

Al igual que las instrucciones anteriores, esta instrucción se compila igual tanto si el código está en CPS como si no.

En la figura 4.10 se muestra un ejemplo de compilación de la instrucción or. Las demás se realizan de modo similar.

```
or destino entero1 entero2
```

```
destino.setValue(((entero1.getValue() != 0) ||
(entero2.getValue() != 0))?1:0);
```

FIGURA 4.10: Compilación de instrucción de or

Como usamos enteros en lugar de booleanos, se deben convertir a `true` o `false`. Seguimos la convención usada en C de que cualquier valor distinto de 0 es `true` y 0 es `false`. Como resultado de la operación se devuelve siempre 0 o 1.

4.3.6.4. Instrucciones de comparación

Este tipo de instrucciones son también independientes de si el código está o no en notación CPS.

En la figura 4.11 se muestra la compilación de la instrucción `greater`. La instrucción `less` se compila de forma similar. En la figura 4.12 se muestra la compilación de la instrucción `equals`.

```
greater destino entero1 entero2
```

```
destino.setValue((entero1.getValue().compareTo(entero2.getValue()) > 0)?1:0);
```

FIGURA 4.11: Compilación de instrucción de "mayor que"

Como se puede observar, las instrucciones `greater` y `less` funcionan para cualquier objeto que implemente la interfaz de Java `Comparable`, al basarse en el método `compareTo`.

```
equals destino entero1 entero2
```

```
destino.setValue((entero1.getValue().equals(entero2.getValue()))?1:0);
```

FIGURA 4.12: Compilación de instrucción de igualdad

La instrucción `equals`, en cambio, se basa en el método `equals` de Java y funciona para cualquier objeto.

4.3.6.5. Instrucción de retorno de función

Un ejemplo de compilación de esta instrucción, tanto a notación usual como a CPS se ha visto y comentado en la sección 4.3.5.

4.3.6.6. Instrucción de llamada a método

Esta instrucción compila a dos versiones diferentes, dependiendo de si se está en un fragmento de código CPS o no. Para ver como se compila esta instrucción, se considerará el ejemplo de la figura 4.13. En este código se define una función que llama a otra función pasando el valor 0 y devuelve el valor retornado por esa función.

```
def ejemplo : Int
      parametro : Int
      retorno : Int

      mov parametro 0

      call retorno <- metodo parametro

      return retorno
end
```

FIGURA 4.13: Ejemplo de instrucción call

En la figura 4.14 se puede observar el código Java generado por el ejemplo anterior para la versión en notación usual del método.

Como se puede observar, la traducción del `call` consiste en llamar a la función correspondiente pasando los parámetros (la referencia, no el valor). El valor retornado por esa llamada a función se asigna al registro destino, en caso de haberlo.

En la figura 4.15, en cambio, se compila el código a notación CPS. La compilación de la instrucción `call` a su versión CPS solo ocurre si existe una versión CPS de la función,

```
public CPS.Reference<Integer> ejemplo(){
      CPS.Reference<Integer> parametro = new CPS.Reference<Integer>();
      CPS.Reference<Integer> retorno = new CPS.Reference<Integer>();
      parametro.setValue((Integer)0);
      retorno.setValue((Integer)metodo(parametro).getValue());
      return retorno;
}
```

FIGURA 4.14: Compilación del ejemplo de instrucción call en notación usual

ya que en caso contrario se trata como una función primitiva y se compila el `call` como en la notación usual, aunque esté dentro de un bloque CPS.

```
public CPS.Reference<?> ejemplo(Function<CPS.Reference<Integer>,
CPS.Reference<?>> CPS_continuacion){
      CPS.Reference<Integer> parametro = new CPS.Reference<Integer>();
      CPS.Reference<Integer> retorno = new CPS.Reference<Integer>();
      parametro.setValue((Integer)0);
      return metodo(
            (CPS_ret1) -> {
                  retorno.setValue((Integer)CPS_ret1.getValue());
                  return CPS_continuacion.apply(retorno);
            }
      , parametro);
}
```

FIGURA 4.15: Compilación del ejemplo de instrucción call en notación CPS

Puede observarse como en este caso la llamada a función se realiza con un argumento adicional (el primero), que provocará que se llame a la versión CPS del método. Este argumento es una lambda de Java 8 conteniendo el código a ejecutar a continuación (la continuación de la instrucción `call`).

Como en este código se guarda el resultado de la función en un registro, esa será la primera instrucción a ejecutar en la continuación. En el caso en que se decidiera ignorar el valor devuelto por la función, esta instrucción no existiría. La segunda instrucción es el return de un registro, que se compila tal y como se vio en la sección 4.3.5.

4.3.6.7. Instrucción de salto condicional (`if`)

La instrucción `if` es otra de las instrucciones que poseen una versión para realizar la compilación en notación usual y otra para cuando se compilan en notación CPS. A continuación consideraremos la compilación del código mostrado en la figura 4.16, que muestra un método que en función de si el valor pasado como parámetro es verdadero o falso devuelve la cadena “Verdadero” o “Falso”.

```
def ejemplo(condicion : Int) : String
      retorno : String

      if condicion {
            mov retorno "Verdadero"
      } else {
            mov retorno "Falso"
      }

      return retorno
end
```

FIGURA 4.16: Ejemplo de instrucción if

Cuando el código se compila en un bloque de código con la notación usual, como el mostrado en la figura 4.17, observamos que el código resultante es básicamente una traducción directa del original, usando las instrucciones `if/else` de Java.

```
public CPS.Reference<String> ejemplo(CPS.Reference<Integer> condicion){
      CPS.Reference<String> retorno = new CPS.Reference<String>();
      if(condicion.getValue() != 0){
            retorno.setValue((String)"Verdadero");
      } else {
            retorno.setValue((String)"Falso");
      }
      return retorno;
}
```

FIGURA 4.17: Compilación del ejemplo de instrucción if en notación usual

En cambio, cuando se compila a notación CPS, como ocurre en la figura 4.18, en lugar de usar `if/else` se invoca a la función `CPS.ifFunction` definida en la biblioteca que se mencionó en la sección 4.3.1. La definición de esta función se puede ver en la figura 4.19.

```
public CPS.Reference<?> ejemplo(Function<CPS.Reference<String>,
CPS.Reference<?>> CPS_continuacion, CPS.Reference<Integer> condicion){
      CPS.Reference<String> retorno = new CPS.Reference<String>();
      return CPS.ifFunction(
            (CPS_ret0) -> {
                  return CPS_continuacion.apply(retorno);
            }
            , condicion,
            () -> {
                  retorno.setValue((String)"Verdadero");
                  return CPS_continuacion.apply(retorno);
            },
            () -> {
                  retorno.setValue((String)"Falso");
                  return CPS_continuacion.apply(retorno);
            }
      );
}
```

FIGURA 4.18: Compilación del ejemplo de instrucción if en notación CPS

Como se puede observar, la función `CPS.ifFunction` acepta tres parámetros, la continuación, el cuerpo de código a ejecutar si se cumple la condición y el cuerpo a ejecutar si no. Como se explicó en la sección 4.3.2, al pasar a CPS se añade la continuación al final de ambas ramas del `if`, por lo que la función ignora la continuación recibida y ejecuta el código de la rama correspondiente, devolviendo el valor que éste retorne. En el ejemplo de la figura 4.18 se observa claramente como el código del `return`, que es la continuación del `if`, se encuentra replicado al final de ambas ramas.

```
static public CPS.Reference<?> ifFunction(
            Function<CPS.Reference<?>, CPS.Reference<?>> CPS_continuation,
            CPS.Reference<Integer> condition,
            Supplier<CPS.Reference<?>> thenBody,
            Supplier<CPS.Reference<?>> elseBody) {
      if (condition.getValue() != 0) {
            return thenBody.get();
      } else {
            return elseBody.get();
      }
}
```

FIGURA 4.19: Definición de la función CPS.ifFunction

4.3.6.8. Instrucción de bucle (`while`)

La instrucción de bucle `while` también se compila de forma distinta para cada tipo de notación. Para ver como se realiza la compilación de esta instrucción, consideraremos el ejemplo mostrado en la figura 4.20.

```
def ejemplo : Int
      numero : Int
      repeticiones : Int
      condicion : Int
      uno : Int

      mov condicion 1
      mov numero 0
      mov repeticiones 10
      mov uno 1

      while condicion {
            call "System.out.println" numero
            add numero numero uno
            less condicion numero repeticiones
      }

      return numero
end
```

FIGURA 4.20: Ejemplo de instrucción while

En este ejemplo, se presenta una función que imprime los números del 0 al 9 (inclusive). Una vez más se observa que la traducción a Java con la notación usual, mostrada en la figura 4.21, es trivial.

```
public CPS.Reference<Integer> ejemplo() {
      CPS.Reference<Integer> numero = new CPS.Reference<Integer>();
      CPS.Reference<Integer> repeticiones = new CPS.Reference<Integer>();
      CPS.Reference<Integer> condicion = new CPS.Reference<Integer>();
      CPS.Reference<Integer> uno = new CPS.Reference<Integer>();
      condicion.setValue((Integer) 1);
      numero.setValue((Integer) 0);
      repeticiones.setValue((Integer) 10);
      uno.setValue((Integer) 1);
      while (condicion.getValue() != 0) {
            System.out.println(numero);
            numero.setValue(numero.getValue() + uno.getValue());
            condicion.setValue((numero.getValue().compareTo(
                        repeticiones.getValue()) < 0) ? 1 : 0);
      }
      return numero;
}
```

FIGURA 4.21: Compilación del ejemplo de instrucción while en notación usual

En cambio el paso a notación CPS, mostrado en la figura 4.18 requiere una explicación, pues no es para nada obvio.

```
public CPS.Reference<?> ejemplo(
            Function<CPS.Reference<Integer>, CPS.Reference<?>> CPS_continuacion)
{
      CPS.Reference<Integer> numero = new CPS.Reference<Integer>();
      CPS.Reference<Integer> repeticiones = new CPS.Reference<Integer>();
      CPS.Reference<Integer> condicion = new CPS.Reference<Integer>();
      CPS.Reference<Integer> uno = new CPS.Reference<Integer>();
      condicion.setValue((Integer) 1);
      numero.setValue((Integer) 0);
      repeticiones.setValue((Integer) 10);
      uno.setValue((Integer) 1);
      CPS.RecursiveCall recCall3 = new CPS.RecursiveCall();
      recCall3.setValue((CPS_retVal3) -> {
            return CPS.whileFunction(
                  (CPS_ret3) -> {
                        return CPS_continuacion.apply(numero);
                  },
                  condicion,
                  () -> {
                        System.out.println(numero);
                        numero.setValue(numero.getValue() + uno.getValue());
                        condicion.setValue((numero.getValue().compareTo(
                                    repeticiones.getValue()) < 0) ? 1 : 0);
                        return recCall3.getValue().apply(null);
                  });
      });
      return recCall3.getValue().apply(null);
}
```

FIGURA 4.22: Compilación del ejemplo de instrucción while en notación CPS

Para ello primero veremos cómo se ha definido la función `CPS.whileFunction` en la biblioteca de la sección 4.3.1. La definición de esta función se muestra en la figura 4.23.

```
static public CPS.Reference<?> whileFunction(
      Function<CPS.Reference<?>, CPS.Reference<?>> cont,
      CPS.Reference<Integer> cond, Supplier<CPS.Reference<?>> body) {
      if (cond.getValue() != 0) {
            return body.get();
      } else {
            return cont.apply(null);
      }
}
```

FIGURA 4.23: Definición de la función CPS.whileFunction

Como se puede ver, la función es sencilla. Acepta tres parámetros, la continuación del while, la condición y el cuerpo del while. Si se cumple la condición, ejecuta el cuerpo del while (cuya última instrucción debería volver a llamar a esta función, para realizar la siguiente iteración). En cambio, si la condición no se cumple se ejecuta la continuación del while (se abandona la ejecución del bucle).

Para ver como implementar el while, hay que observar que si se captura una continuación dentro del cuerpo del while, al terminar de ejecutar el resto de instrucciones del bucle, esta continuación debe volver a evaluar el while, pasando otra vez su cuerpo, el cual contendría otra vez la evaluación del while y así "ad infinitum". Obviamente esto no se puede escribir en una lambda, ya que tendría longitud infinita[3].

La solución que se ha tomado, es crear una lambda que contenga la llamada a la función de while, y guardarla en la estructura de la figura 4.24. Dentro de la propia lambda, al final del cuerpo de la función while se retornará el resultado de invocar esta lamda, implementando la recursión. La razón de que tengamos que guardar la lambda en esta estructura es porque la captura de variables por parte de las lambdas se realiza por valor, y nosotros queremos que la lambda se refiera a sí misma (y eso es imposible al pasarla por valor). Guardándola en esta estructura, es la propia estructura la que se pasa por valor, y no la lambda, por lo que cuando se invoca a la lambda a través de la estructura, se usa el último valor asignado (ella misma).

[3]A este problema se enfrentó también Alonzo Church cuando intentó definir la recursión para el cálculo lambda. Al final lo logró mediante el descubrimiento del combinador Y. Afortunadamente para nosotros, Java es más expresivo que el cálculo lambda y podemos emplear una solución más sencilla.

```
static public class RecursiveCall {
      Function<CPS.Reference<?>, CPS.Reference<?>> value;

      public Function<CPS.Reference<?>,
            CPS.Reference<?>> getValue() {
            return value;
      }

      public void setValue(Function<CPS.Reference<?>,
            CPS.Reference<?>> value) {
            this.value = value;
      }
}
```

FIGURA 4.24: Definición de estructura para pasar la lambda del while por referencia

Por último, se retorna el resultado de invocar a la lambda, que comienza a ejecutar la función while.

4.3.6.9. Instrucción de ámbito de continuación delimitada (`reset`)

La instrucción `reset` tiene tanto una forma en notación usual como una forma CPS (para permitir el anidamiento de instrucciones `reset`). Antes de compilar la instrucción se pasa a notación CPS el cuerpo del `reset`.

En la figura 4.25 se muestra un ejemplo de función en la que aparece una instrucción `reset`. En este ejemplo, dentro del cuerpo de la instrucción `reset` se llama a una función que presumiblemente tiene instrucciones `shift` que devuelven una continuación (objeto de tipo `Continuation`).

```
def ejemplo : Continuation
      retorno : Continuation

      reset retorno{
            call metodo
      }

      return retorno
end
```

FIGURA 4.25: Ejemplo de instrucción reset

En la figura 4.26 se muestra la compilación de la función de ejemplo a la notación usual. Ésta guarda en el registro que se le pasa el resultado de llamar a la función `CPS.resetFunction`, definida en la biblioteca tal y como se muestra en la figura 4.27. Sin embargo, en notación usual se pasa una continuación nula a la función, y ésta retorna

el resultado de ejecutar el cuerpo, que al haber sido transformado a notación CPS será el valor devuelto por un `shift`.

```
public CPS.Reference<CPS.Continuation> ejemplo(){
      CPS.Reference<CPS.Continuation> retorno =
            new CPS.Reference<CPS.Continuation>();
      retorno.setValue((CPS.Continuation)CPS.resetFunction(
            null,
            (CPS_retBody3) -> {
                  return metodo(
                        (CPS_ret4) -> {
                              return new CPS.Reference<Object>();
                        }
                  );
            }
      ).getValue());
      return retorno;
}
```

FIGURA 4.26: Compilación del ejemplo de instrucción reset en notación usual

```
static public CPS.Reference<?> resetFunction(
            Function<CPS.Reference<?>, CPS.Reference<?>> CPS_continuation,
            Function<CPS.Reference<?>, CPS.Reference<?>> body) {
      if (CPS_continuation == null) {
            return body.apply(null);
      } else {
            Reference<?> ret = body.apply(null);
            return CPS_continuation.apply(ret);
      }
}
```

FIGURA 4.27: Definición de la función CPS.resetFunction

En cambio, el resultado de compilar el ejemplo anterior en notación CPS es el mostrado en la figura 4.28. En este caso se retorna el resultado de invocar a la función `CPS.resetFunction`, que será el valor devuelto por un `shift` que se encuentre en la continuación (no el cuerpo) de `CPS.resetFunction`. En este caso se ejecuta el cuerpo del `reset` y el valor retornado por éste (el valor devuelto por un `shift` dentro del cuerpo) se pasa a la continuación (cuya primera instrucción será guardarlo en el registro pasado a `reset`).

4.3.6.10. Instrucción de captura de continuación delimitada (`shift`)

La última instrucción es la instrucción `shift`, que captura la continuación en un registro y ejecuta código que usa ese registro. Esta instrucción no tiene sentido en notación usual, ya que usa la continuación que se le pasa para guardarla en un registro (reificando la continuación). En el ejemplo mostrado en la figura 4.29 puede verse una función que usa

```
public CPS.Reference<?> ejemplo(Function<CPS.Reference<CPS.Continuation>,
      CPS.Reference<?>> CPS_continuacion){
      CPS.Reference<CPS.Continuation> retorno = new
            CPS.Reference<CPS.Continuation>();
      return CPS.resetFunction(
            (CPS_ret8) -> {
                  retorno.setValue((CPS.Continuation)CPS_ret8.getValue());
                  return CPS_continuacion.apply(retorno);
            },
            (CPS_retBody8) -> {
                  return metodo(
                        (CPS_ret9) -> {
                              return new CPS.Reference<Object>();
                        }
                  );
            }
      );
}
```

FIGURA 4.28: Compilación del ejemplo de instrucción reset en notación CPS

la instrucción `shift`. Para que esto tenga sentido, esta función debe llamarse solo dentro de un `reset` (se podría hacer que en notación usual esta instrucción aborte el programa, a modo de excepción sin capturar).

```
def metodo: Int
      entero : Int

      mov entero 0

      shift cont : Continuation {
            return cont
      }

      return entero
end
```

FIGURA 4.29: Ejemplo de instrucción shift

Este ejemplo se compilaría, en notación CPS, al código mostrado en la figura 4.30. En esta función se le pasa a la función `CPS.shiftFunction`, definida en la biblioteca con el código de la figura 4.31, tanto la continuación actual como el cuerpo del `shift` en notación usual. El cuerpo del `shift` acepta como parámetro una referencia a un objeto de tipo `Continuation`.

La función `CPS.shiftFunction` se encarga de copiar la continuación recibida en una nueva referencia a `Continuation` y pasársela al cuerpo de la función. Dentro del cuerpo se debe ejecutar un `return` para dejar de ejecutar el resto del código y salir del `reset` más interno, guardando el valor devuelto en el registro de ese `reset`.

```
public CPS.Reference<?> metodo(Function<CPS.Reference<Integer>,
      CPS.Reference<?>> CPS_continuacion){
      CPS.Reference<Integer> entero = new CPS.Reference<Integer>();
      entero.setValue((Integer)0);
      return CPS.shiftFunction(
            (CPS_ret10) -> {
                  return entero;
            },
            (cont) -> {
                  return cont;
            }
      );
}
```

FIGURA 4.30: Compilación del ejemplo de instrucción shift en notación CPS

```
static public CPS.Reference<?> shiftFunction(
            Function<CPS.Reference<?>, CPS.Reference<?>> CPS_continuation,
            Function<CPS.Reference<Continuation>, CPS.Reference<?>> body) {

      Continuation c = new Continuation(CPS_continuation);
      CPS.Reference<Continuation> continuation = new
            CPS.Reference<Continuation>(c);

      return body.apply(continuation);
}
```

FIGURA 4.31: Definición de la función CPS.shiftFunction

4.3.7. Interoperabilidad con Java

Dentro de éste lenguaje es posible crear registros de algún tipo definido en otra clase Java. Sin embargo no se dispone de instrucción `new` para asignarle valor (aunque no costaría mucho añadirla al lenguaje, ya que es de las que son iguales en cualquier notación). Se le puede asignar valor mediante la llamada a un método de Java que lo haga.

Dentro del lenguaje se puede invocar cualquier método Java que acepte como parámetros referencias a objetos (usando la clase `Reference` definida en la sección 4.3.3). Si se desea guardar el retorno del método, este debe ser también del tipo `Reference`.

Capítulo 5

Ejemplos

En este capítulo veremos algunos ejemplos de cosas que es posible realizar mediante nuestro lenguaje, y que resultan mucho más complejas si el lenguaje no da soporte para ello.

5.1. Continuaciones de escape

En la sección 2.2.2.1 se explicó lo que era una continuación de escape, junto con ejemplos de como implementarla en C o Java.

En la figura 5.1 se muestra una posible implementación usando continuaciones en el lenguaje implementado. La aparente verbosidad del código se debe en su mayor parte a que éste es de bajo nivel y no soporta expresiones.

En este caso se ha decidido realizar algo similar al ejemplo de `setjmp` en C, capturando la continuación al principio y resumiéndola al encontrar un error. Sin embargo, a diferencia de `longjmp` en C, la llamada a la continuación en nuestro lenguaje no retrocede en la pila, por lo que usar esto para escapar de condiciones de error hace que la pila crezca de forma descontrolada.

La salida de este programa es la misma que en los casos anteriormente estudiados, que se vuelve a reproducir a continuación:

```
program Programa

class ContinuacionesEscape

        continuacion : Continuation
        valorLanzado : Int

        def hazAlgo(valor : Int) : Int
                cero : Int
                condicion : Int
                mensajeError : String

                mov cero 0
                equals condicion valor cero
                if condicion {
                        /* Hacer algo */
                } else {
                        mov mensajeError "Detectado error"
                        call "System.out.println" mensajeError
                        mov valorLanzado 1
                        call continuacion.apply
                }
                return cero
        end

        def main : Int
                retShift : Continuation
                cero : Int
                uno : Int
                condicion : Int
                valor : Int
                mensaje : String
                retorno : Int

                mov valorLanzado 0
                mov cero 0
                mov uno 1
                mov valor 0

                reset retShift{
                        shift cont : Continuation {
                                return cont
                        }

                        equals condicion valorLanzado cero
                        if condicion {

                                mov mensaje "Continuación capturada"
                                call "System.out.println" mensaje

                                add valor valor uno
                                call hazAlgo valor

                                return cero

                        } else {
                                mov mensaje "Continuación invocada"
                                call "System.out.println" mensaje

                                mov mensaje "El valor es "
                                call "System.out.print" mensaje
                                call "System.out.println" valor

                                return uno
                        }
                }

                mov continuacion retShift
                call retorno <- continuacion.apply
                return retorno
        end
end
```

FIGURA 5.1: Ejemplo de emulación de continuaciones de escape en nuestro lenguaje

```
Continuación capturada
Detectado error
Continuación invocada
El valor es 1
```

Además, es posible realizar las continuaciones de escape de forma similar a como se realizan las excepciones. En este caso no usamos las continuaciones reificadas, sino que nos aprovechamos de que la instrucción `shift` descarta la continuación delimitada actual. Por tanto, podemos emplear `reset` a modo de `try` y `shift` a modo de `throw` para obtener un código con los mismos resultados, como el que se muestra en la figura 5.2.

La salida de este código es exactamente igual a la del mostrado anteriormente, pero a diferencia del anterior la pila no crece con cada uso, por lo que podría considerarse como una implementación de excepciones que no requiere de las excepciones de Java.

5.2. Generadores

Otro tipo de estructura que es sencillo implementar mediante continuaciones son los generadores. Como ya se ha explicado, estos son elementos parecidos a los iteradores pero que, en lugar de recorrer una colección finita generan de forma perezosa el siguiente elemento de una colección potencialmente ilimitada. En la figura 5.3 se muestra lo sencillo que es hacer un generador que cada vez que se invoque genere el siguiente elemento de la secuencia de los cuadrados de los números naturales.

Como se puede observar, el uso de las continuaciones permite programar el generador como si se tratara de un bucle usual, pero retornando cada valor mediante un `shift`, de forma que se calculen solo los números que se desean.

Además, no es necesario indicar qué información debe guardarse para calcular el siguiente número, ya que la continuación lleva implícita toda la información necesaria.

La salida de este código será, por tanto:

```
0
1
4
9
16
```

```
program Programa

class ContinuacionesEscape

	def hazAlgo(valor : Int) : Int
		cero : Int
		uno : Int
		condicion : Int
		mensajeError : String

		mov cero 0
		mov uno 1
		equals condicion valor cero
		if condicion {
			mov cero 0
		} else {
			mov mensajeError "Detectado error"
			call "System.out.println" mensajeError
			shift cont : Continuation {
				return uno
			}
		}
		return cero
	end

	def main : Int
		valorLanzado : Int
		cero : Int
		uno : Int
		condicion : Int
		valor : Int
		mensaje : String

		mov valorLanzado 0
		mov cero 0
		mov uno 1
		mov valor 0

		reset valorLanzado {

			mov mensaje "Continuación capturada"
			call "System.out.println" mensaje

			add valor valor uno
			call hazAlgo valor
		}

		equals condicion uno valorLanzado
		if condicion {
			mov mensaje "Continuación invocada"
			call "System.out.println" mensaje

			mov mensaje "El valor es "
			call "System.out.print" mensaje
			call "System.out.println" valor

			return uno
		}
		else
		{
			return cero
		}
	end
end
```

FIGURA 5.2: Segundo ejemplo de emulación de continuaciones de escape en nuestro lenguaje

```
program myProgram

class GeneraCuadrados
      def main : Int
            continuacion : Continuation
            valor : Int

            call getGenerador continuacion

            call valor <- continuacion.apply
            call "System.out.println" valor

            call valor <- continuacion.apply
            call "System.out.println" valor

            call valor <- continuacion.apply
            call "System.out.println" valor

            call valor <- continuacion.apply
            call "System.out.println" valor

            call valor <- continuacion.apply
            call "System.out.println" valor

            return valor
      end

      def getGenerador(continuacion : Continuation) : Int
            numero : Int
            resultado : Int
            uno : Int
            retorno : Int

            mov uno 1
            mov numero 0

            reset retorno {

                  while uno {
                        shift cont : Continuation {
                              mov continuacion cont
                              return resultado
                        }
                        mov resultado numero
                        mul resultado resultado resultado
                        add numero numero uno
                  }
            }

            return uno
      end
end
```

FIGURA 5.3: Generador de números cuadrados

Como resultado de las ventajas anteriores, usando continuaciones todos los generadores se realizan de forma muy parecida, solo cambiando las reglas para generar el siguiente elemento. Por ejemplo en la figura 5.4, cambiando aproximadamente tres líneas de código con respecto al ejemplo anterior se ha programado un generador de números de la secuencia de Fibonacci.

Para ello solo se ha modificado la forma de generar el siguiente número, que en lugar de ser un iterador multiplicado por sí mismo pasa a ser la suma de los dos anteriores. Es interesante comparar este generador con el equivalente en Python, mostrado en la figura 2.9.

La salida de este código se muestra a continuación:

```
1
1
2
3
5
```

```
program myProgram

class GeneraFibonacci
      def main : Int
            continuacion : Continuation
            valor : Int

            call getGenerador continuacion

            call valor <- continuacion.apply
            call "System.out.println" valor

            call valor <- continuacion.apply
            call "System.out.println" valor

            call valor <- continuacion.apply
            call "System.out.println" valor

            call valor <- continuacion.apply
            call "System.out.println" valor

            call valor <- continuacion.apply
            call "System.out.println" valor

            return valor
      end

      def getGenerador(continuacion : Continuation) : Int
            numero1 : Int
            numero2 : Int
            resultado : Int
            uno : Int
            retorno : Int

            mov uno 1
            mov numero1 1
            mov numero2 1

            reset retorno {

                  while uno {
                        shift cont : Continuation {
                              mov continuacion cont
                              return resultado
                        }
                        mov resultado numero1
                        mov numero1 numero2
                        add numero2 resultado numero2
                  }
            }

            return uno
      end
end
```

Figura 5.4: Generador de números de Fibonacci

Capítulo 6

Conclusiones

En este trabajo se han estudiado las continuaciones y las continuaciones delimitadas.

Se ha implementado un lenguaje de bajo nivel que da soporte a continuaciones, se ejecuta sobre la JVM y es capaz de interoperar con Java. Este lenguaje se ha desarrollado mediante una aproximación estática, consistente en realizar un paso parcial a notación CPS. Esto puede ser realizado sobre cualquier otro lenguaje de alto nivel para dotarle de continuaciones, sin que requiera soporte especial del lenguaje.

Se han explicado detalladamente los aspectos técnicos de esta implementación, con el fin de facilitar a futuros desarrolladores la realización de proyectos semejantes compilando a cualquier otro lenguaje, y se han propuesto ejemplos de construcciones realizables gracias a estas funcionalidades.

6.1. Posibles desarrollos futuros

Para finalizar se indican posibles direcciones en las que podría expandirse el material de este proyecto, las cuales no se han realizado por quedar fuera del alcance del proyecto original.

Podría realizarse un estudio de las construcciones posibles mediante este lenguaje y una comparativa con otros lenguajes con construcciones similares.

Al ser un lenguaje que solo pretendía mostrar la implementación de continuaciones sobre este lenguaje, existen multitud de instrucciones y elementos que no se han introducido

al lenguaje, como puede ser la definición de funciones anónimas o la creación de clases anidadas. Sería posible extender el lenguaje para que soporte más características y estudiar si esto afecta a la implementación de las continuaciones.

Alternativamente se podría intentar crear un lenguaje de más alto nivel que soporte expresiones y compile a este lenguaje, con el fin de comprobar las posibles dificultades que se puedan encontrar y estudiar su viabilidad como lenguaje de desarrollo de aplicaciones.

Bibliografía

[1] Raúl Rojas. A tutorial introduction to the lambda calculus. *arXiv preprint arXiv:1503.09060*, 2015.

[2] Java 8 lambdas - a peek under the hood. http://www.infoq.com/articles/Java-8-Lambdas-A-Peek-Under-the-Hood. Accedido: 11-06-2015.

[3] Richard Warburton et al. *Java 8 Lambdas: Pragmatic Functional Programming*. .º'Reilly Media, Inc.", 2014.

[4] *The Open Group Base Specifications Issue 7 - setjmp*. The Open Group, 2001-2013. *Disponible en http://pubs.opengroup.org/onlinepubs/9699919799/functions/setjmp.html*.

[5] Iulian Dragos, Antonio Cunei, and Jan Vitek. Continuations in the java virtual machine. In *Second ECOOP Workshop on Implementation, Compilation, Optimization of Object-Oriented Languages, Programs and Systems (ICOOOLPS"2007)*, number LAMP-CONF-2007-002. Technische Universität Berlin, 2007.

[6] Lukas Stadler, Christian Wimmer, Thomas Würthinger, Hanspeter Mössenböck, and John Rose. Lazy continuations for java virtual machines. In *Proceedings of the 7th International Conference on Principles and Practice of Programming in Java*, pages 143–152. ACM, 2009.

[7] Jvm continuations. http://wiki.jvmlangsummit.com/images/2/2b/JVMLanguageSummit_Stadler_Continuations.pdf. Accedido: 16-06-2015.

[8] Tiark Rompf, Ingo Maier, and Martin Odersky. Implementing first-class polymorphic delimited continuations by a type-directed selective cps-transform. *ACM Sigplan Notices*, 44(9):317–328, 2009.

[9] Bill Venners. *Inside the Java virtual machine.* McGraw-Hill, Inc., 1996.

[10] Joshua Engel. *Programming for the Java virtual machine.* Addison-Wesley Professional, 1999.