Programación funcional para el resto de nosotros

Slava Akhmechet

29 de agosto de 2010

Resumen

Un excelente punto de partida para comprender mejor la programación funcional. Original de Slava Akhmechet y traducido al español por Luis Mendoza con el permiso del autor. En la dirección electrónica *defmacro.org/ramblings/fp.html* puedes consultar el original.

1. Introducción

Los programadores son procrastinadores (o sea, personas que aplazan las cosas). Llegan, toman un poco de café, revisan su bandeja de entrada, leen sus actualizaciones de RSS, leen las noticias, dan un vistazo a los artículos más recientes en los sitios de tecnología, examinan las discusiones políticas en los secciones designadas de los foros de programación. . . se restriegan los ojos y echan otro vistazo para asegurarse de no perderse nada. Van a comer. Regresan, inician el IDE por unos minutos. Revisan la bandeja de entrada. Toman un poco de café. Antes de darse cuenta, el día de trabajo ya terminó.

Pero, de vez en cuando, te encuentras con artículos verdaderamente desafiantes. Si buscas en el lugar correcto encontrarás al menos uno cada pocos días. Como son difíciles de entender y necesitas tiempo para leerlos, empiezan a acumularse. Antes de darte cuenta, tienes una larga lista de vínculos y una carpeta llena de archivos PDF y quisieras tener un año en una pequeña cabaña a la mitad del bosque sin nadie a kilómetros a la redonda para que finalmente puedas comprenderlos. Estaría bien que alguien viniera cada mañana mientras das un paseo por el río para dejarte algo de comida y llevarse la basura.

No sé de tu lista, pero una buena parte de la mía tiene que ver con programación funcional. Estos son generalmente los artículos más difíciles de entender. Escritos en un áspero lenguaje académico, aun los "veteranos de la industria de Wall Street con diez años de experiencia" no entienden de qué tratan los artículos sobre programación funcional (también llamada FP, por sus siglas en inglés). Si preguntas al administrador de proyectos de Citi Group o de Deutsche Bank por qué usan JMS en lugar de Erlang te dirán que no pueden usar lenguajes académicos para aplicaciones de fortaleza industrial [1]. El problema es que algunos de los sistemas más complejos con los requerimientos más rígidos están escritos usando elementos de programación funcional. Algo no cuadra.

Es cierto que los artículos sobre FP (recuerda que son las siglas en inglés para programación funcional) son difíciles de entender, pero no tendrían por qué serlo. Las razones para que haya ese obstáculo al conocimiento son meramente históricas. No hay nada inherentemente difícil en los conceptos de FP. Considera este artículo como "una guía accesible para entener FP", un puente entre nuestras mentes imperativas hacia el mundo de FP. Prepárate un café y sigue leyendo. Con un poco de suerte, en poco tus amigos estarán bromeando sobre tus nuevas ideas sobre FP.

Así que, ¿qué es la programación funcional o FP? ¿Cómo se produce? ¿Es comestible? Si es tan útil como proclaman sus defensores, ¿por qué no es más usada en la industria? ¿Por qué parece

[1] Cuando buscaba trabajo en el otoño de 2005 a menudo hice esa pregunta. Es bastante divertido recordar cuantos rostros con signos de interrogación vi. Uno podría pensar que a $300,000 la pieza de estas personas, al menos tendrían un buen entendimiento de las herramientas que tienen disponibles.

que solo personas con grado de doctorado lo quieren usar? Más importante, ¿por qué es tan difícil de aprender? ¿Qué es todo eso de cierres(closures), continuaciones, currying, evaluación tardía y cero efectos colaterales? ¿Cómo puede usarse en proyectos que no tengan que ver son universidades? ¿Por qué parece tan distinto de todo lo bueno, santo y querido por nuestros corazones imperativos? Aclararemos todo esto pronto. Empecemos explicando las razones por las que existe esa gran barrera entre el mundo real y los artículos académicos. La respuesta es tan sencilla como dar un paseo por el parque.

2. Un paseo por el parque

¡Enciende la máquina del tiempo! Nuestro paseo empieza unos dos mil años atrás, en una bello y soleado día de la primavera de 380 a.C. A las afueras de las murallas de Atenas, bajo la plácida sombra de los olivos, Platón caminaba rumbo a la Academia con un joven pupilo. El clima estaba agradable, la cena había estado deliciosa, y la conversación empezó a tomar tintes filosóficos.

"Mira a esos dos estudiantes", dijo Platón escogiendo cuidadosamente las palabras para hacer una pregunta educativa. "¿Cual de ellos te parece más alto?" El joven pupilo miró hacia la cuenca de agua en la que dos hombres estaban parados. "Son más o menos de la misma altura", contestó. Platón preguntó: "¿Qué quieres decir con 'más o menos'?". El joven contesto: "Bueno, desde aquí se ven de la misma estatura, pero si estuviera más cerca seguramente vería alguna diferencia".

Platón sonrió. Había llevado al joven en la dirección correcta. "¿Así que dirías que no hay ninguna cosa perfectamente igual a otra en nuestro mundo?" Después de pensar un poco, el joven respondió: "No lo creo. Toda cosa es al menos un poco diferente de otra, aunque no podamos ver la diferencia". ¡Había dado en el clavo! Platón dijo: "Entonces, si ninguna cosa es perfectamente igual a otra en este mundo, ¿cómo crees que entendemos el concepto de equidad 'perfecta'?" El joven se quedó perplejo. "No lo sé", contestó.

Así nacio el primer esfuerzo por entender la naturaleza de las matemáticas. Platón sugirió que todo en nuestro mundo es solo una aproximación de la perfección. Además, se dio cuenta de que entendemos el concepto de perfección aunque no la hayamos visto. Llegó a la conclusión de que las formas matemáticas perfectas viven en otro mundo y que de alguna forma sabemos de ellas al tener una conexión con ese universo "alternativo". Sabemos bien que no hay un círculo perfecto que podamos observar. Pero entendemos qué es un círculo perfecto y podemos describirlo mediante ecuaciones. ¿Qué son, entonces, las matemáticas? ¿Por qué esta descrito nuestro universo con leyes matemáticas? ¿Será posible que todos los fenómenos del universo sean descritos mediante las matemáticas?[2]

La filosofía de las matemáticas es una materia de estudio muy compleja. Como muchas disciplinas filosóficas genera más preguntas que respuestas. Buena parte de los concensos alcanzados giran en torno a que las matemáticas son realmente un rompecabezas: podemos declaramos un conjunto básico de principios que no entran en conflicto, y un conjunto de reglas sobre cómo operan estos principios. Entonces podemos usar estas reglas como base para reglas más y más complejas. Los matemáticos le llaman a esto un "sistema formal" o "cálculo". Podríamos construir un sistema formal del Tetris si quisieramos. De hecho, una implementación del Tetris que funcione *es* un sistema formal, solo que especificado usando una representación inusual.

Una civilización de criaturas peludas que existiera en Alfa Centauri no podría leer nuestros formalismos del Tetris y de los círculos porque su único órgano sensorial solo percibiera olores. Lo más probable es que nunca sabrían nada del Tetris, pero sí tendrían un formalismo para los círculos. Probablemente nosotros no podríamos entenderlo, pues nuestro sentido del olfato no sería tan sofisticado, pero una vez que dejamos de lado la *representación* del formalismo (a través de

[2]Esta parece ser una cuestión muy controversial. Los físicos y los matemáticos se ven obligados a reconocer que no esta del todo claro si todo en el universo obedece leyes matemáticas o no.

diversos instrumentos sensoriales y técnicas de lectura para entender el *lenguaje*), los conceptos fundamentales son entendibles para cualquier civilización inteligente.

Es interesante notar que aunque no existiera una civilización inteligente en el universo, los formalismos del Tetris y de los círculos estarían ahí para ser examinados, solo que nadie los estaría buscando. Si una civilización inteligente apareciese, es muy probable que descubriría algunos formalismos para ayudarle a describir las leyes de nuestro universo. Sería poco probable que alguna vez encontraran algo sobre el Tetris porque nada en el universo observable se le parece. El Tetris es uno de los incontables ejemplos de sistemas formales, o rompecabezas, que no tienen nada que ver con el mundo real. De hecho, no podemos estar seguros de que todos los números naturales tengan referencia en el mundo real, ya que podemos pensar en un número tan grande que no pueda describir nada en un universo donde las cosas tienden a ser finitas.

3. Un poco de historia[3]

Cambiemos de velocidad en la máquina del tiempo. Esta vez viajaremos a un punto más cercano, a la decada de 1930. La Gran Depresión estaba asolando a todo el mundo. Casi toda familia de toda clase social se vió afectada por la tremenda recesión económica. Muy pocos santuarios quedaron donde la gente estaba a salvo de la pobreza. Pocas personas tuvieron la fortuna de estar en alguno de esos santuarios, pero existían. Nuestro interés se centrará en los matemáticos de la Universidad de Princeton.

Las nuevas oficinas construidas en estilo gótico daban a Princeton un aire de paraíso. Lógicos (de la rama matemática de la lógica) de todo el mundo fueron invitados a Princeton para construir un nuevo departamento. Mientras muchos en norteamérica no podían ni poner una pieza de pan en sus mesas para la cena, techos altos, paredes cubiertas de madera tallada, discusiones diarias con una taza de té, y paseos por el bosque eran algunas de las condiciones de Princeton.

Uno de los matemáticos viviendo ese lujoso estilo de vida fue un joven llamado Alonzo Church. Alonzo recibió un grado académico en Princeton, y fue persuadido a quedarse para un postgrado. Alonzo sentía que la arquitectura del lugar era más un lujo que algo necesario. Rara vez se le veía discutiendo sobre matemáticas con una taza de té, y no le gustaban los paseos por el bosque. Alonzo era solitario. Era más productivo cuando trabajaba por su cuenta. No obstante, Alonzo tenía contacto regular con otros habitantes de Princeton, entre ellos Alan Turing, John von Neumann, y Kurt Gödel.

Los cuatro estaban interesados en los sistemas formales. No prestaban mucha atención al mundo físico; les resultaban más interesantes problemas con rompecabezas matemáticos abstractos. Sus rompecabezas tenían algo en común: los cuatro estaban interesados en responder preguntas sobre *computación*. Si tuvieramos máquinas con infinito poder computacional, ¿Que problemas podríamos resolver? ¿Podrían darse soluciones automáticamente? ¿Quedarían algunos problemas sin resolver? ¿Por qué? ¿Podrían maquinas con diferentes diseños ser iguales en poder?

En cooperación con otros, Alonzo desarrolló un sistema formal llamado *cálculo lambda*. El sistema era esencialmente un lenguaje de programación para una de esas máquinas imaginarias. Se basaba en funciones que tomaban otras funciones como parámetros, y que devolvían funciones como resultado. La función en su papel de materia prima fue identificada con la letra griega lambda(Λ), de ahí el nombre del sistema [4]. Usando este formalismo, Alonzo pudo razonar sobre muchas de las

[3]Siempre he odiado las lecciones de historia que presentan cronológicamente fechas, nombres y eventos de forma mecánica. Para mí, la historia es sobre las vidas de las personas que cambiaron el mundo. Es sobre sus razones personales que les llevaron a actuar de cierta forma, y los mecanismos por medio de los cuales cambiaron millones de vidas. Por esta razón, esta sección histórica es irremediablemente incompleta. Solo las personas y los eventos importantes son discutidos.

[4]Cuando estaba aprendiendo programación funcional me quedé muy confundido por el término "lamda" porque no sabía qué significaba realmente. En este contexto lambda es una función. El uso de la letra griega es porque era más fácil escribirla en notación matemática. Cada vez que escuches el término "lambda" cuando se habla de programación funcional traducelo en tu mente como "función".

cuestiones antes planteadas, y llegó a respuestas concluyentes.

Independientemente de Alonzo, Alan Turing realizó un trabajo similar. Desarrolló un formalismo diferente (ahora conocido como la máquina de Turing), y lo usó para llegar a conclusiones similares a las de Alonzo. Después se demostró que la máquina de Turing y el cálculo lamda son equivalentes en poder.

Aquí es donde la historia terminaría. Yo terminaría el artículo, y tu navegarías a otra página, si no fuera por el comienzo de la Segunda Guerra Mundial. El mundo estaba encendido. La armada y la marina estadounidenses usaban artillería más que nunca. En un esfuerzo por mejorar la precisión de artillería, el ejercito empleó a un gran grupo de matemáticos que continuamente calculaban las ecuaciones diferenciales requeridas para resolver tablas de balística. Se fue haciendo evidente que la tarea era demasiado complicada para resolverla manualmente, y se desarrollaron varios equipos para solucionar el problema. La primer máquina para resolver tablas de balística fue Mark I, construida por IBM. Pesaba cinco toneladas, tenía 750,000 partes y podía hacer hasta tres operaciones por segundo.

La carrera, por supuesto, no había terminado. En 1949 fue presentada la computadora EDVAC y tuvo un éxito tremendo. Fue el primer ejemplo de la arquitectura de von Neumann y fue efectivamente una implementación en el mundo real de la máquina de Turing. Alonzo Church no tuvo mucha suerte aquella vez.

A finales de la década de 1950, el profesor del MIT John McCarthy (también graduado de Princeton) se interesó en el trabajo de Alonzo Church. En 1958 presentó el lenguaje de procesamiento de listas (Lisp). ¡Lisp era una implementación del cálculo lambda que trabajaba en computadoras von Neumann! Muchos científicos en computación reconocieron el poder expresivo de Lisp. En 1973, un grupo de programadores del Laboratorio de Inteligencia Artificial del MIT desarrollaron hardware al que llamaron "máquina Lisp", ¡una implementación nativa en hardware del cálculo lambda de Alonzo!

4. Programación funcional

La programación funcional es la implementación práctica de las ideas de Alonzo Church. No todas las ideas del cálculo lambda se implementan en la práctica porque el cálculo lambda no fue diseñado para trabajar bajo limitaciones físicas. Por tanto, como en la programación orientada a objetos, la programación funcional es un conjunto de ideas, no un conjunto estricto de reglas. Hay muchos lenguajes de programación funcional, y la mayoría hacen las cosas de formas muy diferentes entre sí. En este artículo explicaré las ideas más ampliamente usadas en los lenguajes funcionales usando ejemplos tomados del lenguaje Java (sí, puedes escribir programas funcionales en Java si eres masoquista). En las siguientes secciones tomaremos Java tal cual, y le haremos modificaciones para transformarlo en lenguaje funcional útil. Empecemos nuestro viaje.

El cálculo lambda fue creado para investigar problemas relacionados con *cálculo*. La programación funcional, por tanto, trata principalmente con cálculo, y, ¡sorpresa!, lo hace mediante funciones. La función es la unidad básica en programación funcional. Las funciones son usadas para prácticamente todo, aun para los cálculos más simples. Hasta las variables son reemplazadas con funciones. En programación funcional las variables son simplemente accesos directos a expresiones (para no tener que escribir todo en una misma línea). No pueden ser modificadas. Todas las variables pueden ser asignadas solo una vez. En términos de Java esto significa que cada variable es declarada como *final* (o *const* si hablamos de C++). No hay variables mutables en FP (ve el Listado 1).

Listado 1: Todas las variables son finales

```
final int i = 5;
final int j = i + 3;
```

Dado que cada variable en FP es final, podemos llegar a dos conclusiones interesantes. Primero, que no tiene sentido escribir la palabra clave *final* y segundo, que no tiene sentido llamar a las variables, pues... variables. Haremos estas dos modificaciones a Java: cada variable declarada en nuestro Java funcional sera final por default, y nos referiremos a las variables como *símbolos*.

Para ahora probablemente te estas preguntando cómo podrías escribir algo razonablemente complicado en nuestro lenguaje recién creado. ¡Si todo símbolo es inmutable, entonces no podemos cambiar el estado de nada! Esto no es estrictamente cierto. Cuando Alonzo estaba trabajando en el cálculo lambda no estaba interesado en mantener un estado para ser modificado posteriormente. Estaba interesado en realizar operaciones sobre datos (tambien conocidos como "material de cálculo"). Como sea, el cálculo lambda es equivalente en poder a la máquina de Turing. Un lenguaje funcional puede hacer lo mismo que un lenguaje imperativo. ¿Cómo, entonces, podemos obtener los mismos resultados?

En realidad los programas funcionales pueden mantener un estado, pero no usan las variables para eso. Usan funciones. El estado es mantenido en los parámetros de la función, en la pila. Si deseas mantener un estado para modificarlo posteriormente, escribes una función recursiva. Como ejemplo, escribamos una función que devuelva una cadena de carácteres de Java al revés. Recuerda que cada variable (más bien, cada símbolo) es final por default[5].

Listado 2: Función que devuelve una cadena al revés

```
String al_reves(String arg) {
   if (arg.length() == 0) {
      return arg;
   } else {
      return al_reves(
         arg.substring(1, arg.length()) +
         arg.substring(0, 1));
   }
}
```

Esta función es lenta porque se llama a sí misma repetidamente [6]. Es una devoradora de memoria porque repetidamente asigna memoria a los objetos. Pero está escrita en estilo funcional. Quizá te preguntes porque alguien querría un programa de esta forma. Bueno, estaba a punto de decírtelo.

5. Beneficios de FP

Probablemente pienses que no hay forma en que yo pueda defender la monstruosidad de función mostrada arriba. Cuando estaba aprendiendo programación funcional pensaba igual. Pero estaba equivocado. Hay muy buenos argumentos para usar este estilo de programación. Algunos son subjetivos. Por ejemplo, algunos defensores de la programación funcional arguyen que son más entendibles. Dejaré de lado esos argumentos porque todos sabemos que la facilidad para entender algo es completamente subjetivo. Afortunadamente, hay una buena cantidad de argumentos objetivos.

5.1. Prueba de unidades

Dado que todo símbolo en FP es final, ninguna función puede causar efectos colaterales. No puedes modificar los símbolos que ya tienen un valor, y ninguna función puede modificar un valor fuera de su ámbito para ser usado por otra función (a diferencia de un miembro de clase o una variable global). Eso significa que el único efecto de evaluar una función es su valor de retorno y que la única cosa que afecta el valor de retorno de la función son sus argumentos.

[5] Es interesante notar que las cadenas de Java son inmutables de todas formas. Es aun más interesante explorar las razones de esta falsedad, pero nos distraeríamos de nuestro objetivo.

[6] Muchos de los compiladores de lenguajes funcionales optimizan las funciones recursivas transformandolas en sus contrapartes iterativas siempre que sea posible. A eso se le llama optimización de llamadas a la inversa (*Tail recursion*).

Eso es el sueño de todo probador de unidades. Puedes probar cada función de tu programa preocupandote unicamente por sus argumentos. No hay que preocuparse por llamar las funciones en un orden correcto, o configurar apropiadamente un escenario basado en estados externos. Todo lo que necesitas es pasar a la función los argumentos que representan los casos de prueba. Si cada función en tu programa pasa las pruebas de unidad puedes tener más confianza sobre la calidad de tu software que si estas usando un lenguaje imperativo. En Java o C++ revisar el valor de retorno de una función no es suficiente (esto es porque la función pudo haber modificado el estado externo, que también debe verificarse). No así en un lenguaje funcional.

5.2. Depuración

Si un programa funcional no trabaja como se esperaría, depurarlo es cosa fácil. El problema se repetirá siempre, pues un programa funcional no depende de lo que ocurra antes o después en otra parte que no sea la función misma. En un programa imperativo, un problema puede aparecer algunas veces y otras no. Dado que las funciones en un programa imperativo dependen del estado externo producido por efectos colaterales de otras funciones, quizá tengas que revisar subrutinas de código que no están relacionadas directamente con el problema. No así en un programa funcional. Si el valor de retorno de una función es erróneo, entonces *siempre* será erróneo, sin importar el código ejecutado antes o después de la función en cuestión.

Una vez reproducido el problema, llegar al fondo del asunto es trivial, es casi placentero. Detienes la ejecución del programa y examinas la pila. Cada argumento en la llamada a una función está en la pila disponible para su inspección, igual que en un programa imperativo. Excepto que en un programa imperativo esto no es suficiente, pues además hay que revisar variables miembro, variables globales y el estado de otras clases (que a su vez pueden depender de más y más cosas). Una función en un programa funcional depende *únicamente* de sus argumentos, ¡y esa información esta justo frente a tus ojos! Más aun, en un programa imperativo examinar el valor de retorno de una función no te dará una idea completa de si la función trabaja correctamente. Debes revisar docenas de objetos en el exterior de la función para verificar que hizo su trabajo bien. ¡En un programa funcional lo único que tienes que revisar es el valor de retorno!

Al examinar la pila verás los argumentos pasados a las funciones y sus valores de retorno. En el momento en que algún valor de retorno no tenga sentido, pasas a examinar la función implicada. ¡Repites este proceso recursivamente hasta llegar a la fuente del problema!

5.3. Concurrencia

Un programa funcional está listo para ejecución concurrente sin ningun tipo de adaptación. No necesitas preocuparte por condiciones de carrera o bloqueos mutuos, ¡porque no usas bloqueos! Si ninguna pieza de datos en un programa funcional es modificada dos veces en el mismo hilo de ejecución, mucho menos en hilos diferentes. Eso significa que puedes agregar hilos a tu aplicación, ¡libre de los problemas que plagan a las aplicaciones concurrentes en los programas imperativos!

Si este es el caso, ¿por qué nadie usa programas funcionales para aplicaciones altamente concurrentes? Bueno, a decir verdad, sí lo hacen. Ericsson diseño un lenguaje funcional llamado Erlang para ser usado en conmutadores de telecomunicaciones escalables y altamente tolerantes a fallos. Muchos más han reconocido los beneficios que ofrece Erlang y han comenzado a usarlo. Estamos hablando de sistemas de telecomunicaciones y control de tráfico que son, por mucho, más confiables y escalables que los sistemas típicamente diseñados para Wall Street. A decir verdad, los sistemas Erlang no son escalables ni confiables. Los sistemas Java sí. Los sistemas Erlang son simplemente roca sólida.

La historia de la concurrencia no termina aquí. Si tu aplicación es inheréntemente de un solo hilo, el compilador puede aun optimizar programas funcionales para correr en múltiples CPUs. Hecha un vistazo al siguiente fragmento de código (Listado 3):

Listado 3: Funciones potencialmente paralelizables

```
String s1 = unaOperacionMuyLarga1();
String s2 = unaOperacionMuyLarga2();
String s3 = concatenar(s1, s2);
```

En un lenguaje funcional el compilador puede analizar el código, clasificar las funciones que crean las cadenas *s1* y *s2* como potenciales consumidoras de tiempo, y ejecutarlas concurrentemente. Esto es imposible de hacer en un lenguaje imperativo porque cada función puede modificar el estado externo y la siguiente función a ejecutarse quizá dependa de ello. En los lenguajes funcionales el análisis automático de funciones para encontrar buenos candidatos para ejecución concurrente es tan trivial como reducir funciones a funciones *inline* en C++. ¡Así de fácil! En este sentido, los programas en estilo funcional son "a prueba del futuro" (por mucho que odie las expresiones de moda, esta vez hare una excepción). Los creadores de hardware ya no pueden hacer que las CPUs sean más rápidas. En su lugar, incrementan el número de núcleos y atribuyen el cuádruple de aumento de velocidada a la concurrencia. Por supuesto, convenientemente olvidan mencionar que solo obtendremos beneficios de eso si el software que usamos esta listo para ser paralelizable. Esto apenas es cierto en una pequeña fracción del software imperativo, pero es verdad en el 100 % del software funcional, porque los programas funcionales son paralelizables recién salidos de la caja.

5.4. Actualizaciones en caliente(*Hot Code Deployment*)

Hace tiempo, para instalar actualizaciones de Windows era necesario reiniciar la computadora. Varias veces. Y eso solo para instalar una nueva versión del reproductor multimedia. Con Windows XP la situación ha mejorado bastante, pero aun esta lejos de ser ideal (hoy corrí Windows Update en el trabajo y ahora un molesto icono en la bandeja del sistema no desaparecerá hasta que reinicie). Los sistemas Unix han tenido un mejor modelo desde hace tiempo. Para instalar una actualización, solo necesitas detener los componentes relevantes, no el sistema operativo entero. Aunque esto es mejor, para cierto tipo de aplicaciones de servidor aun es inaceptable. Los sistemas de telecomunicaciones necesitan estar al 100 % todo el tiempo, pues si por una actualización del sistema no se puede realizar una llamada de emergencia algunas vidas podrías perderse. No hay razón para que las firmas de Wall Street deban detener sus sistemas para actualizaciones durante el fin de semana.

Una situación ideal sería actualizar las partes importantes del código sin detener ninguna parte del sistema. En un mundo imperativo, eso sería imposible. Piensa en lo que implicaría reemplazar una clase Java en tiempo de ejecución. Si lo hicieramos, entonces toda instancia de dicha clase se volvería inutil porque el estado que encapsula se perdería. Tendríamos que escribir código de control de versiones muy complicado para manejar la situación. Primero, tendrían que serializarse todas las instancias de clase en ejecución, destruirlas, crear nuevas instancias con la nueva definición de la clase e intentar recargar los datos serializados en estas nuevas instancias, esperando que los datos migrados efectivamente funcionen en las nuevas instancias. Encima de todo, cada vez que hagamos cambios en la definición de una clase, tendríamos que escribir el código de migración manualmente. Y el código de migración necesitaría prestar cuidadosa antención para no romper las relaciones entre los objetos. Tal vez suene bien en teoría, pero nunca funcionaría bien en la práctica.

En un programa funcional todo el estado está guardado en la pila, en los argumentos pasados a las funciones. ¡Esto hace que las actualizaciones en caliente sean significativamente más fáciles! De hecho, todo lo que necesitamos hacer es ejecutar *diff* entre el código en producción y la nueva versión, y entonces actualizar con el código nuevo. El resto puede ser hecho por las herramientas de lenguaje, ¡automáticamente! Si crees que esto es ciencia ficción, piénsalo de nuevo. Los ingenieros de Erlang han estado actualizando sistemas sin detenerlos, y eso ya por varios años.

5.5. Pruebas y optimizaciones asistidas por computadora

Una característica interesante de los lenguajes funcionales es que se puede razonar sobre ellos matemáticamente. Dado que un lenguaje funcional es simplemente una implementación de un sistema formal, todas las operaciones matemáticas que pueden hacerse sobre papel también aplicarán a los programas escritos en dicho lenguaje. El compilador puede, por ejemplo, convertir piezas de código en equivalentes más eficientes con la comprobación matemática de que ambas piezas de código son equivalentes[7]. Las bases de datos relacionales han usado este tipo de optimizaciones por años. No hay razón por la que estas mismas técnicas no se usen en otro tipo de software.

Adicionalmente, puedes usar estas técnicas para probar que ciertas partes de tu programa son correctas. ¡Hasta es posible crear herramientas que analicen el código y generen casos de pruebas para comprobación de unidades automáticamente! Esta funcionalidad es invaluable para sistemas sólidos como las rocas. Si estás diseñando marcapasos o sistemas de control de tráfico aéreo estas herramientas son casi siempre un requerimiento. Si estas escribiendo aplicaciones que no pertenezcan a las verdaderas industrias de misión crítica, de todas formas estas herramientas te darán una tremenda ventaja sobre tus competidores.

6. Funciones de órden superior

Recuerdo que cuando estaba aprendiendo sobre estos beneficios de los que he hablado pensaba: "eso es muy bonito, pero inutil si tengo que escribir en un lenguaje lisiado donde todo es final". Pero estaba en un error. Hacer todas las variables finales *es* lisiarlas en el contexto de un lenguaje imperativo como Java, pero no en el contexto de los lenguajes funcionales. Los lenguajes funcionales ofrecen un tipo diferente de herramientas de abstracción que te haran olvidar siquiera *haber querido* modificar variables. Una de esas herramientas es la capacidad de trabajar con *funciones de orden superior*.

Una función en un lenguaje funcional es diferente de una función en C o en Java. Es un superconjunto (es decir, puede hacer todo lo que una función Java puede hacer, y más). Crearemos una función de la misma manera que lo hacemos en C (Listado 4.)

Listado 4: Función que suma dos números

```
int suma(int i, int j) { return i + j; }
```

Sin embargo, este trozo de código difiere del equivalente en C. Extendamos nuestro compilador Java para soportar esta nueva notación de la que hablo. Cuando escribamos algo como lo anterior, nuestro compilador lo convertirá en el fragmento de código Java del Listado 5 (no olvides que todo es final):

Listado 5: Función expandida en nuestro Java funcional

```
class t_funcion_suma{
   int suma(int i, int j) {
      return i + j;
   }
}

t_funcion_suma suma = new t_funcion_suma();
```

El símbolo *suma* no es realmente una función. Es la instancia de una pequeña clase con una sola función miembro. Por eso, podemos pasar *suma* en nuestro código como un argumento para otras

[7] Lo contrario no siempre es verdad. Mientras que a veces es posible probar que dos piezas de código son equivalentes, esto no es posible en todas las situaciones.

funciones. Podemos asignarla a otro símbolo. Podemos crear instancias de *t_funcion_suma* en tiempo de ejecución y serán eliminadas por el recolector de basura cuando ya no se necesiten. Esto hace de las funciones *objetos de primera clase*, no distintos de los enteros o las cadenas. Las funciones que operan sobre otras funciones (o sea, que aceptan funciones como argumentos) son llamadas *funciones de órden superior (higher order functions)*. No dejes que dicho término te intimide. No es diferente de las clases Java que operan sobre otras (podemos pasar instancias de una clase a otra). Podríamos llamarlas "clases de orden superior", pero a nadie le importa porque no hay una fuerte comunidad académica detrás de Java.

¿Cómo y cuando usaremos funciones de orden superior? Que bueno que preguntes. Cuando empiezas un programa como un aglutinado de código monolítico sin preocuparte por jerarquía de clases, y luego te das cuenta de que una pieza particular de código se repite muchas veces, entonces la conviertes en una función (afortunadamente aun enseñan esto en las escuelas). Si ves que una pieza de lógica dentro de tu función necesita comportarse diferente en diferentes situaciones, entonces la conviertes en una función de orden superior. ¿Confundido? He aquí un ejemplo de la vida real tomado de mi trabajo.

Supon que tenemos la pieza de código Java del Listado 6 que recibe un mensaje, lo transforma de varias maneras, y lo reenvía a otro servidor.

Listado 6: Controlador de mensajes para un solo servidor

```
class ControladorMensajes {
   void controlarMensaje(Mensaje msg) {
      // ...
      msg.configurarCodigoCliente("ABCD_123");
      // ...

      enviarMensaje(msg);
   }

   // ..
}
```

Ahora imagina que nuestro sistema ha cambiado y que tenemos que encaminar mensajes a dos servidores en lugar de uno. Todo es manejado de la misma forma a excepción del código del cliente (el segundo servidor lo quiere en un formato diferente). ¿Cómo abordamos la situación? Revisaríamos donde es enviado el mensaje y elegiríamos entre los códigos del cliente para elegir el correcto, como se muestra en el Listado 7:

Listado 7: Controlador de mensajes para dos servidores

```
class ControladorMensajes {
   void controlarMensaje(Mensaje msg) {
      // ...
      if (msg.obtenerDestino().equals("server1") {
         msg.configurarCodigoCliente("ABCD_123");
      } else {
         msg.configurarCodigoCliente("123_ABCD");
      }
      // ...

      enviarMensaje(msg);
   }

   // ..
}
```

Sin embargo esta solución no es escalable. Si se agregaran más servidores nuestra función crecería linealmente y actualizarla sería una pesadilla. Una solución de enfoque orientado a objetos es hacer

ControladorMensajes una clase base y especializarla con el código de cada cliente en las clases derivadas (ve el Listado 8):

Listado 8: Controlador de mensajes orientado a objetos

```
abstract class ControladorMensajes {
   void controlarMensaje(Mensaje msg) {
      // ...
      msg.configurarCodigoCliente(obtenerCodigoCliente());
      // ...

      enviarMensaje(msg);
   }

   abstract String obtenerCodigoCliente();

   // ..
}

class ControladorMensajes1 extends ControladorMensajes {
   String obtenerCodigoCliente() {
      return "ABCD_123";
   }
}

class ControladorMensajes2 extends ControladorMensajes {
   String obtenerCodigoCliente() {
      return "123_ABCD";
   }
}
```

Ahora podemos instanciar una clase apropiada para cada servidor. Añadir más servidores es ahora más escalable. Aun así, es mucho código para una modificación tan simple. ¡Tuvimos que crear dos nuevos tipos solo para soportar distintos tipos de códigos de clientes! Ahora, hagamos la misma cosa en nuestro lenguaje que soporta funciones de orden superior (Listado 9):

Listado 9: Controlador de mensajes con funciones de órden superior

```
class ControladorMensajes {
   void controlarMensaje(Mensaje msj,
                         Function codigoCliente)
   {
      // ...
      Mensaje msj1 = msg.configurarCodigoCliente(codigoCliente());
      // ...

      enviarMensaje(msj1);
   }

   // ...
}

String codigoCliente1() {
   return "ABCD_123";
}

String codigoCliente2() {
   return "123_ABCD";
}

ControladorMensajes controlador = new ControladorMensajes();
controlador.controlarMensaje(unMsj, codigoCliente1);
```

En este caso, no creamos tipos nuevos, ni tuvimos que lidiar con jerarquías de clases. Simplemente pasamos la función apropiada como parámetro. Conseguimos lo mismo que con la contraparte orientada a objetos, pero con varias ventajas adicionales. No nos restringimos a jerarquías de clases: pordemos pasar nuevas funciones en tiempo de ejecución y cambiarlas en cualquier momento con un mayor grado de granularidad y menos código. ¡El compilador ha escrito código orientado a objetos "aglutinante" para nosotros! Además, tenemos todos los otros beneficios de FP. Por supuesto, las abstracciones provistas por los lenguajes funcionales no terminan aquí. Las funciones de órden superior son solo el principio.

7. *Currying*

Mucha gente que conozco ha leído el libro Patrones de Diseño (*Design Patterns*) escrito por La Pandilla de los Cuatro. Todo programador que se precie te dirá que el libro es agnóstico respecto al lenguaje de implementación y que los patrones aplican a la ingeniería de software en general, sin importar el lenguaje que uses. Esa es una declaración noble. Desafortunadamente está lejos de la realidad.

Los lenguajes funcionales son extremadamente expresivos. En un lenguaje funcional uno no necesita los patrones de diseño porque el lenguaje es de tan alto nivel, que terminas programando en conceptos que los hacen innecesarios. Uno de esos patrones es el patrón de Adaptador (¿Cómo se diferencia del patron *Facade*? Suena como si alguién necesitara llenar más páginas para satisfacer su contrato). Este es innecesario en un lenguaje que soporta una técnica llamada *currying*.

El patrón de Adaptador es mejor conocido cuando se aplica a la unidad de abstracción por *default* en Java (una clase). En lenguajes funcionales este patrón es aplicado a funciones. Dicho patrón toma una interfaz y la transforma en otra interfaz que alguien más espera. Aquí hay un ejemplo de un patrón de Adaptador (Listado 10):

Listado 10: Ejemplo de un patron de adaptación

```
int potencia(int i, int j);
int cuadrado(int i)
{
   return potencia(i, 2);
}
```

El código arriba mostrado adapta una interfaz de una función que eleva un entero a un exponente entero, a una interfaz de la función que eleva al cuadrado un entero. En círculos académicos esta técnica trivial es llamada *currying* (en honor a Haskell Curry, quien realizó las acrobacias matemáticas necesarias para formalizar esto). Dado que las funciones en FP son pasadas como argumentos (opuesto a las clases), *currying* es usado a menudo para adaptar funciones a una interfaz que alguien más espera. Dado que la interfaz de las funciones son sus argumentos, la técnica *currying* es usada para reducir el número de argumentos (como en el ejemplo anterior).

Los lenguajes funcionales vienen con esta técnica incorporada. No necesitas crear manualmente una función que envuelva a la otra, el lenguaje mismo lo hará por ti. Como antes, extendamos nuestro lenguaje para incorporar esta técnica.

Listado 11: *Currying* automático a una función

```
cuadrado = int potencia(int i, 2);
```

El código del Listado 11 creará por nosotros una función llamada *cuadrado* con un solo argumento. Llamará a la función *potencia* con el segundo argumento establecido a 2. Esto compilará en el siguiente código Java (Listado 12):

Listado 12: Expansión de *currying* en nuestro Java funcional

```
class t_funcion_cuadrado {
   int cuadrado(int i) {
      return potencia(i, 2);
   }
}

t_funcion_cuadrado cuadrado = new t_funcion_cuadrado();
```

Como puedes ver, de forma simple creamos una envoltura de la función original. En FP *currying* es solo eso (una forma fácil de y rápida de crear envolturas de otras funciones). Tu te concentras en tu trabajo, ¡y el compilador escribe el código apropiado para ti! ¿Cuando usar currying? Es fácil de responder. Cada vez que quieras usar el patrón de Adaptador (una envoltura).

8. Evaluación tardía

La evaluación tardía (o perezosa) es un técnica interesantes que se vuelve posible una vez que adoptamos una filosofía funcional. Ya habíamos visto el fragmento de código del Listado 13 cuando hablamos sobre concurrencia:

Listado 13: Fragmento de código candidato a evaluación tardía

```
String s1 = unaOperacionMuyLarga1();
String s2 = unaOperacionMuyLarga2();
String s3 = concatenar(s1, s2);
```

En un lenguaje imperativo el orden de evaluación queda claro. Dado que cada función pudiera afectar o depender del estado externo, es necesario ejecutarlas en órden: primero *unaOperacionMuyLarga1*, luego *unaOperacionMuyLarga2*, seguida de *concatenar*. No así en un lenguaje funcional.

Como vimos antes, *unaOperacionMuyLarga1* y *unaOperacionMuyLarga2* pueden ser ejecutadas concurrentemente porque está garantizado que ninguna función afecta o depende del estado global. Pero si no es necesario ejecutarlas concurrentemente, ¿necesitamos ejecutarlas en orden? La respuesta es no. Solo necesitamos ejecutar estas operaciones cuando otra función dependa de *s1* y *s2*. Ni siquiera tenemos que ejecutarlas antes de *concatenar*. Si reemplazamos *concatenar* con una función que tenga una condición y use solo uno de sus parámetros, ¡quizá nunca evaluemos el otro parámetro! Haskell es un ejemplo de lenguaje de evaluación tardía. En Haskell no hay garantía de que el código sea ejecutado en órden (o que siquiera sea ejecutado), pues Haskell solo ejecuta el código cuando es requerido.

La evaluación tardía tiene numerosas ventajas así como desventajas. Discutiremos las ventajas aquí, y veremos cómo lidiar con las desventajas en la siguiente sección.

8.1. Optimización

La evaluación tardía ofrece un tremendo potencial para optimizaciones. Un compilador de este tipo ve al código funcional exactamente como un matemático ve una expresión algebraica. Puede cancelar cosas y prevenir completamente su ejecución, reacomodar las piezas para mayor eficiencia, incluso reacomodar el código de una forma que reduzca los errores, todo esto garantizando que la lógica permanecerá intacta. Este es el mayor beneficio de representar programas usando estrictamente primitivas formales: cómo el código se adhiere a leyes matemáticas, se puede pensar en él matemáticamente.

8.2. Abstracción de estructuras de control

La evaluación tardía provee un nivel superior de abstracción que permite implementar cosas que de otra forma serían imposibles. Por ejemplo, considere implentar la estructura de control del Listado 14:

Listado 14: Estructura de control personalizada

```
aMenosQue(stock.esEuropeo()) {
   enviarASEC(stock);
}
```

Deseamos ejecutar *eviarASEC* a menos que el stock sea europeo. ¿Cómo podríamos implementar *aMenosQue*? Sin evaluación tardía, necesitariamos alguna tipo de macro, pero en un lenguaje como Haskell no. ¡Podemos implementar *aMenosQue* como una función! (Ve el Listado 15)

Listado 15: Implementación de la estructura de control con evaluación tardía

```
void aMenosQue(boolean condicion, List codigo) {
   if (!condicion)
      codigo;
}
```

Nota que *codigo* nunca es evaluado si la condición es verdadera. No podemos hacer los mismo en un lenguaje de evaluación estricta porque los argumentos serían evaluados antes de entrar en *aMenosQue*.

8.3. Estructuras de datos infinitas

Los lenguajes de evaluación tardía permiten la definición de estructuras de datos infinitas, algo que es mucho más complicado en un lenguaje de evaluación estricta. Por ejemplo, considera una lista con los números de Fibonacci. Está claro que no podemos realizar cálculos sobre una lista infinita en un tiempo razonable, o guardarla en memoria. En lenguajes de evaluación estricta como Java, simplemente definimos una function *fibonacci* que devuelva un miembro particular de la secuencia. En un lenguaje como Haskell podemos abstraerlo aun más y simplemente definir una lista infinita con los números de Fibonacci. Como el lenguaje es de evaluación tardía, solo las partes necesarias de la lista que son usadas realmente por el programa serán evaluadas. Esto permite abstraer muchos problemas y verlos desde una persepectiva de más alto nivel (por ejemplo, podemos usar procesamiento de listas sobre una lista infinita).

8.4. Desventajas

Por supuesto, no todo es miel sobre hojuelas. La evaluación tardía implica también desventajas. La principial es que, pues, es tardía (o perezosa). Muchos de los problemas del mundo real requieren evaluación estricta. Por ejemplo, considera el Listado 16:

Listado 16: Código que requiere evaluación estricta

```
System.out.println("Por favor ingresa tu nombre: ");
System.in.readLine();
```

¡En un lenguaje de evaluación tardía no hay garantía de que la primera línea se ejecutará antes de la segunda! Esto significa que no podemos hacer operaciones de E/S, ni usar funciones nativas de ninguna forma util (puesto que necesitan ser llamadas en órden, dado que dependen de efectos colaterales), ¡ni interactuar con el mundo exterior! Si hicieramos que las primitivas fueran ejecutadas en un orden específico, perderíamos los beneficios de poder ver nuestro código desde un punto de vista matemático (con lo que se irían también todos los beneficios de la programación funcional).

Afortunadamente, no todo está perdido. Los matemáticos han trabajado y desarrollado algunos trucos para asegurarse de que porciones del código sean ejecutadas en un órden particular en un ambiente funcional. ¡Tenemos lo mejor de los dos mundos! Estas técnicas incluyen continuaciones, *monads*, y escritura singular. En este artículo solo veremos las continuaciones. Dejaremos las otras dos para otra ocasión. Es interesante que las continuaciones son útiles para más cosas que permitir un órden particular de evaluación. Hablaremos de esto también.

9. Continuaciones

Las continuaciones son a la programación lo que el Código Da Vinci a la historia de la humanidad: la revelación de uno de los más grandes encubrimientos conocidos por el hombre. Bueno, quizá no tanto, pero ciertamente es una revelación del engaño en la misma forma que las raíces cuadradas de números negativos.

Cuando aprendimos sobre funciones, solo aprendimos la verdad a medias, pues asumimos fálsamente que las funciones deben regresar su valor de retorno a quien la llamó. En este sentido, las continuaciones son una generalización de las funciones. Una función no necesita regresar a quien la llamó, y más bien puede ir a cualquier otra parte del programa. Una "continuación" es un parámetro opcional que le dice a la función donde debe continuar la ejecución del programa. La descripción quizá suene más complicada de lo que parece. Mira el fragmento de código del Listado 17:

Listado 17: Dos líneas de código dependientes

```
int i = suma(5, 10);
int j = cuadrado(i);
```

La función *suma* devuelve *15*, valor que es asignado al símbolo *i*, en el lugar donde *suma* fue llamado. Después el valor de *i* es usado para llamar *cuadrado*. Note que un compilador de evaluación tardía no puede reacomodar estas líneas de código pues la segunda línea depende de la evaluación correcta de la primera. Podemos reescribir este bloque de código usando el *Estilo de Pase de Continuación* o *CPS* (por las siglas en inglés para *Continuation Pass Style*), donde la función *suma* no regresa a quien la llamó sino entrega su resultado a la función *cuadrado* (Listado 18).

Listado 18: Estilo de pase de continuación (*CPS*)

```
int j = suma(5, 10, cuadrado);
```

En este caso a *suma* se le pasa otro parámetro, una función a la que *suma* debe llamar cuando haya terminado su trabajo. En este caso, *cuadrado* es la continuación de *suma*. En ambos casos (Listados 17 y 18) *j* será igual a 225.

Aquí encontramos el primer truco para forzar a nuestro lenguaje de evaluación tardía a evaluar dos expresiones en orden. Considera el fragmento de código de E/S del Listado 19 (muy familiar por cierto):

Listado 19: E/S que depende del órden de ejecución

```
System.out.println("Por favor ingresa tu nombre: ");
System.in.readLine();
```

Estas dos líneas no dependen una de la otra, por lo que el compilador tiene libertad para reacomodarlas como desee. Sin embargo, si reescribimos esto en *CPS* (*Estilo de Pase de Continuación*), ¡entonces habrá una dependencia y el compilador se verá forzado a evaluar las dos líneas en órden! (Listado 20)

Listado 20: E/S escrita en estilo de pase de continuación (*CPS*)

```
System.out.println("Por favor ingresa tu nombre: ", System.in.readLine);
```

En este caso *println* necesita llamar a *readLine* con su resultado, y devolver el resultado de *readLine*. Esto nos permite asegurarnos de que de que las dos líneas serán ejecutadas en el órden deseado *y* que *readLine* será evaluada (porque toda la expresión depende del último valor como resultado). En el caso de la función *println* devuelve *void*, pero si devolviera cualquier valor abstracto que *readLine* acepte, ¡entonces nuestro problema queda solucionado! Por supuesto, encadenar funciones de esta forma se vuelve pronto en algo ilegible, pero no tiene por qué ser así. Podemos añadir un poco de *azúcar sintáctica* al lenguaje, de tal forma que nos permita simplemente escribir las expresiones en orden, y que el compilador encadene las llamadas por nosotros automáticamente. ¡Ahora podemos evaluar las expresiones en el órden que queramos sin perder ninguno de los beneficios de FP (incluyendo al habilidad de razonar sobre nuestro programas matemáticamente)! Si aun te resulta confuso, recuerda que una función es solo una instancia de una clase con una solo método. Reescribe las dos líneas de tal forma que *println* y *readLine* sean instancias de clases y lo entenderás más claramente.

En este momento yo cerraría esta sección, si no fuera porque solo hemos arañado la superficie de las continuaciones y su utilidad. Podemos escribir programas enteros en estilo *CPS*, donde cada función reciba como parámetro adicional una continuación (la función donde el programa debe continuar) y entregar el resultado a esa función de continuación. Además podemos convertir cualquier programa a *CPS* simplemente al tratar las funciones como casos especiales de continuaciones (funciones que siempre regresan a quien las llamó). Esta conversión es trivial y puede efectuarse automáticamente (de hecho, muchos compiladores lo hacen).

Una vez que un programa es convertido al estilo *CPS* queda claro que cada instrucción tiene un tipo de continuación, una función a la que llamará con su resultado, que en un programa regular sería el punto donde debe retornar. Tomemos una instrucción del código anterior, digamos *suma(5, 10)*. En un programa en estilo *CPS* esta claro cual es la continuación de *suma* (es la función a la que llama *suma* cuando ha terminado). Pero, ¿que hay en el caso de un programa que no está escrito en estilo *CPS*? Podríamos, por supuesto, convertir el programa a *CPS*, pero, ¿es necesario?

Resulta que no. Mira con cuidado a nuestra conversión *CPS*. Si trataras de escribir un compilador para esto y piensas lo suficiente en ello, te darás cuenta de que un programa en estilo *CPS*, ¡no necesita pila para las llamadas a funciones! Ninguna función "regresa" en el sentido tradicional, sino que simplemente llama a otra función. No necesitamos empujar los argumentos de la función en la pila con cada llamada para luego sacarlos otra vez. Podemos simplemente guardarlos en un bloque de memoria y usar la instrucción *jump* en su lugar. No necesitaremos más los argumentos originales (¡nunca son usados de nuevo pues ninguna función "regresa"!).

Así que los programas en estilo *CPS* no necesitan pila sino un argumento extra con la función a la cual llamar a continuación. Los programas que no están escritos en estilo *CPS* no tiene el argumento de continuación extra, sino la pila. ¿Que contiene la pila en esos casos? Simplemente los argumentos, y un puntero a donde la función debe retornar. ¿Se te prendió el foco? ¡La pila contiene únicamente información de continuación! ¡El puntero para la instrucción de retorno en la pila es esencialmente lo mismo que la función a llamar en programas *CPS*! Si deseas saber cual es la continuación para *add(5, 10)*, ¡simplemente examinas la pila en ese punto!

Así de simple. Una continuación y un puntero para la instrucción de retorno son la misma cosa, solo que la continuación es pasada explícitamente, de tal forma que no necesita ser el mismo lugar desde donde la función fue llamada. Si recuerdas que una continuación es una función, y que una función en nuestro lenguaje es compilado a una instancia de una clase, te darás cuenta de que un puntero para la instrucción de retorno y el argumento de continuación son *realmente* la misma cosa, dado que nuestra función (al igual que una instancia de clase) es simplemente un puntero. Esto significa que en algun punto de tu programa puedes averiguar cuál es la *continuación actual* (que es simplemente la información que se encuentra en la pila).

Bien, ahora ya sabemos lo que es una continuación actual. ¿De que nos sirve? Cuando obtenemos la continuación actual y la guardamos en algun lugar, en realidad estamos guardando el estado actual de nuestro programa (como si lo congelaramos en el tiempo). Esto es similar a cuando el sistema

operativo entra en hibernación. Un *objeto de continuación* contiene la información necesaria para reiniciar el programa desde el punto donde fue creado. Un sistema operativo hace esto todo el tiempo cuando cambia de contexto entre los hilos. La única diferencia es que el sistema operativo lo controla todo. Si tu solicitas un objeto de continuación (en el lenguaje Scheme se hace mediane llamar a la función *call-with-current-continuation*) obtienes un objeto que contiene la continuación actual (toda la información en la pila, o la siguiente función a ser llamada en el caso de *CPS*). Ahora puedes guardar este objeto en una variable (o en el disco). Cuando deseas "reiniciar" tu programa con este objeto de continuación, "transformas" el estado de tu programa al que está guardado en el objeto de continuación. Es parecido a regresar a un hilo suspendido o despertar a una computadora de hibernación, a excepción de que lo puedes una y otra y otra vez. Cuando un sistema operativo despierta, la información de hibernación es destruida. Si no fuera así, podrías despertarla una y otra vez en el mismo punto, como si volvieras en el tiempo. ¡Tienes esa clase de control con las continuaciones!

¿En que situaciones resultan útiles las continuaciones? Usualmente cuando tratas de simular estado en aplicación que inherentemente no tienen estado. Una gran aplicación de las continuaciones son las aplicaciones web. ASP.NET de Microsoft hace un esfuerzo enorme para tratar de simular estado en aplicaciones web de tal forma que puedas escribir aplicaciones sin tanto lío. Si C# soportara continuaciones, la mitad de la complejidad de ASP.NET desaparecería (solo guardarías un objeto de continuación y lo reiniciarías cuando el usuario hiciera una nueva petición al servidor web). Desde el punto de vista del programador de la aplicación web, no habría interrupción (¡el programa simplemente continuaría en la siguiente línea!). Las continuaciones son una abstracción increíblemente útil para algunos problemas. Considerando que muchos de los pesados cliente tradicionales se están moviendo a la web, las continuaciones cobrarán mayor relevancia en el futuro.

10. Coincidencia de patrones (*Pattern Matching*)

La coincidencia de patrones no es una característica inovadora. De hecho, tiene poco que ver con la programación funcional. La única razón por la que usualmente se le atribuye a FP es porque los lenguajes funcionales han tenido coincidencia de patrones por algun tiempo, mientras que los lenguajes imperativos modernos aun no.

Echemos un vistazo a la coincidencia de patrones mediante un ejemplo. El listado 21 muestra la función Fibonacci en Java:

Listado 21: Función *Fibonacci* en Java

```
int fib(int n) {
   if (n == 0) return 1;
   if (n == 1) return 1;

   return fib(n-2) + fin(n-1);
}
```

El Listado 22 muestra la función Fibonacci en nuestro lenguaje derivado de Java con soporte para coincidencia de patrones.

¿Cual es la diferencia? Que el compilador implementa la bifurcación por nosotros. ¿Cual es la gran cosa con esto? Ninguna. Alguien notó que un gran número de funciones contienen muchas sentencias *switch* complicadas (esto es particularmente cierto en los programas funcionales) y decidió que sería una buena idea abstraerlo de esta forma. Dividimos la función en muchas, y ponemos patrones en lugar de algunos de los argumentos (algo parecido a la sobrecarga de funciones). Cuando la función es llamada, el compilador compara los argumentos con las definiciones en tiempo de ejecución y elige la correcta. Esto se hace usualmente eligiendo la definición más específica. Por ejemplo, *int fib(int n)* podría ser llamada siendo *n* igual a *1*, pero en este caso *int fib(1)* es más específica.

Listado 22: Función *Fibonacci* con conincidencia de patrones

```
int fib(0) {
   return 1;
}

int fib(1) {
   return 1;
}

int fib(int n) {
   return fib(n-2) + fib(n-1);
}
```

La coincidencia de patrones es usualmente más compleja de la que se observa en estos ejemplos. Por ejemplo, un sistema avanzado de coincidencia de patrones nos permitiría hacer lo siguiente (Listado 23):

Listado 23: Patrones de coincidencia avanzados

```
int f(int n < 10){ ... }
int f(int n) { ... }
```

¿Cuando resulta útil usar coincidencia de patrones? ¡En un gran número de situaciones! Cada vez que tengas una compleja estructura de *ifs* anidados, la coincidencia de patrones puede hacer un mejor trabajo con menos código de tu parte. Una función que viene a la mente es la función estándar *WndProc* que todas las aplicaciones Win32 deben proveer (y esto a menudo se hace mediante abstracciones). Usualmente un sistema de coincidencia de patrones puede examinar tanto colecciones como valores simples. Por ejemplo, si pasas un arreglo (*array*) a tu función, podrías elegir entre los arreglos cuyo primer elemento sea *1* y el tercero sea mayor que *3*.

Otro beneficio de la coincidencia de patrones es que si necesitas añadir o modificar condiciones, no tienes que lidiar con una única función enorme. Simplemente añades (o modificas) las definiciones apropiadas. Esto elimina la necesidad de un buen número de los patrones de diseño descritos en el libro de la Pandilla de los Cuatro. Mientras más complejas sean las condiciones, más te será útil la coincidencia de patrones. Una vez que te acostumbras, empiezas a preguntarte cómo pudiste vivir tanto tiempo sin esto.

11. Cierres (*Closures*)

Hasta ahora hemos discutido características de lenguajes funcionales "puros" (lenguajes que implementan el cálculo lambda y que no entran en conflicto con los formalismos de Church). Sin embargo, muchas de las características de los lenguajes funcionales son útiles fuera del ámbito del cálculo lambda. Mientras que la implemetación de un sistema axiomático es util pues nos permite pensar en nuestros programas desde el punto de vista matemático, quizá esto no siempre sea práctico. Muchos de los lenguajes eligen incorporar algunos elementos funcionales sin adherirse estrictamente a la doctrina funcional. Muchos de esos lenguajes (como Common Lisp) no requieren que todas las variables sean finales (puedes modificarlas en cualquier momento). Además, no requieren que las funciones dependan sólo de sus argumentos (las funciones pueden accesar al estado externo). Pero incluyen características funcionales (como las funciones de orden superior). El paso de funciones en un lenguaje "impuro" es un poco diferente que hacerlo en los confines del cálculo lambda, y precisa una interesante característica a veces llamada cierre léxico. Veamos el código de ejemplo del Listado 24. Recuerda, en este caso las variables no son finales y las funciones pueden referirse a variables fuera de su ámbito:

Listado 24: Ejemplo de cierre léxico

```
Function crearFuncionPotencia(int potencia) {
   int funcionPotencia(int base) {
      return elevar(base, potencia);
   }

   return funcionPotencia;
}

Funcion cuadrado = hacerFuncionPotencia(2);
cuadrado(3) ; //Devuelve 9
```

La función *crearFuncionPotencia* devuelve una función que toma un único argumento y que lo eleva a cierta potencia. ¿Que sucede cuando evaluamos *cuadrado(3)*? La variable *potencia* ya no está en el ámbito de *funcionPotencia* porque *crearFuncionPotencia* ya ha terminado y lo que había en la pila ya no está. ¿Cómo puede funcionar *cuadrado* entonces? El lenguaje debe, de alguna forma, guardar el valor de *potencia* para que *cuadrado* funcione. ¿Que pasaría si crearamos otra función llamada *cubo* que elevara a la tercera potencia? En tiempo de ejecución existen dos copias de *potencia*, una por cada función generada usando *crearFuncionPotencia*. Al fenómeno de guardar estos valores se les llama *cierres*. Los cierres no guardan solo argumentos de la función anfitriona. Por ejemplo, un cierre puede tomar esta forma (Listado 25):

Listado 25: Incrementador usando cierres

```
Function crearIncrementador() {
   int n = 0;

   int incremento() {
      return ++n;
   }
}

Function inc1 = crearIncrementador();
Function inc2 = crearIncrementador();

inc1(); // devuelve 1
inc1(); // devuelve 2
inc1(); // devuelve 3
inc2(); // devuelve 1
inc2(); // devuelve 2
inc2(); // devuelve 3
```

En tiempo de ejecución se guarda el valor de *n*, para que los incrementadores puedan accesarlo. Más que eso, guarda varias copias de *n*, una por cada incrementador, aunque se suponga que desaparecerían cuando *crearIncrementador* retorne. ¿En que se convierte este código al compilar? ¿Cómo funcionan los cierres detrás del telón? Afortunadamente, tenemos un pase para los camerinos.

Un poco de sentido común nos ayudará. La primera observación que hacemos es que las variables locales no están limitadas a reglas simples de resolución de ámbito y pueden tener un tiempo de vida indeterminado. La conclusión obvia es que no están guardadas en la pila (más bien están guardadas en el montículo[8]). Un cierre, entonces, es implementado justo como cualquiera de las funciones antes vistas, a excepción de que tiene referencias adicionales a las variables en el ámbito de su anfitrinón (Ve el Listado 26).

Cuando un cierre hace referencia a una variable que no está en el ámbito local, entonces consulta esta referencia en el ámbito del padre (o función anfitriona). ¡Eso es todo! Los cierres hacen que los mundos funcional y orientado a objetos estén más cerca. Cada vez que creas una clase que mantiene

[8]Esto no es más lento que guardas valores en la pila, pues una vez que introduces el recolector de basura, la asignación de memoria se vuelve una operación O(1).

Listado 26: Tabla de símbolos del ámbito superior

```
class t_alguna_funcion {
   SymbolTable ambientePadre;

   // ...
}
```

algún estado y lo pasa a algun otro lugar, piensa en los cierres. Un cierre es solamente un objeto que crea "variables miembro" al vuelo al tomarlas de su ámbito, ¡para que no tengas que hacerlo tu!

12. ¿Que sigue?

Este artículo solo toca la superficie de la programación funcional. A veces un pequeño esbozo puede convertirse en algo grande, y en nuestro caso eso es algo muy bueno. En el futuro planeo escribir sobre teoría de categorías, *monads*, estructuras de datos funcionales, sistemas de escritura en lenguajes funcionales, concurrencia funcional, bases de datos funcionales y mucho más. Si logro escribir (y en el proceso, aprender) sobre la mitad de estos temas, mi vida estará completa. Mientras tanto, Google es nuestro amigo.

13. ¿Tienes algun comentario?

Si tienes alguna pregunta, comentario, o sugerencia, por favor déjame una nota en coffeemug@gmail.com. Me dará gusto saber tus opiniones.