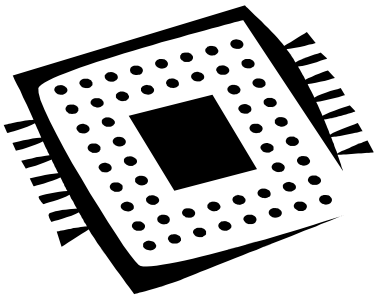
INSTITUTO TECNOLÓGICO Y DE ESTUDIOS
SUPERIORES DE MONTERREY

CAMPUS ESTADO DE MÉXICO

TECNOLÓGICO
DE MONTERREY®



Programación multinúcleo

Artículos de investigación sobre tecnologías y lenguajes de programación concurrentes y/o paralelos.

Editor: Prof. Ariel Ortiz Ramírez

Diciembre, 2012.

Introducción

Este documento es un compendio de trece trabajos de investigación elaborados por alumnos de la carrera de Ingeniero en Sistemas Computacionales (ISC) para la materia *Tc3035 Programación multinúcleo* ofrecida durante el semestre de agosto-diciembre del 2012. Esta es la primera vez que se imparte este curso en el Campus Estado de México del Tecnológico de Monterrey. La materia corresponde a una optativa profesional para el plan de ISC 2009. Los alumnos la pueden cursar en cualquiera de los últimos tres semestres de la carrera.

El objetivo de la materia es que los alumnos conozcan y apliquen las metodologías de programación y las herramientas para análisis de rendimiento diseñadas para lograr el funcionamiento más eficiente de sus programas en ambientes de cómputo basados en procesadores de múltiples núcleos y de procesamiento concurrente. Los trabajos que aquí se presentan buscan complementar el material que se cubrió en clase.

Cada uno de estos trabajos fue elaborado de manera individual o en parejas. El contenido de los artículos se enfoca en los aspectos concurrentes y/o paralelos de la tecnología o lenguaje en cuestión, aunque también incluyen una introducción a aspectos más generales con el fin de proveer un mejor contexto. Los temas específicos fueron asignados a cada equipo a través de un sorteo. Los textos fueron compuestos usando el sistema de preparación de documentos LaTeX.

El lector de esta obra deberá juzgar la calidad de cada artículo de manera individual, pero como editor puedo decir que quedé muy satisfecho del resultado global.

Profesor Ariel Ortiz Ramírez
7 de diciembre, 2012.

Tabla de contenido

Ada, el lenguaje de programación 1

El lenguaje de programación paralelo Chapel 7

Cilk para un C más facil 15

Concurrencia en Curry 22

Concurrencia en D 29

Lenguaje de programación Fortress y paralelismo 38

Programación multinúcleo utilizando F# 46

Go, el lenguaje de programación de Google 56

Capacidades concurrentes del lenguaje Io 61

Concurrencia en Modula-3 69

OpenCL, programación concurrente y paralela 75

El lenguaje multiparadigma Oz 85

Scala: Un lenguaje scalable 95

Ada, el lenguaje de programación

Jorge Manuel Ramos Peña (A00904604)
Instituto Tecnológico y de Estudios Superiores de Monterrey
Campus Estado de México
Atizapán de Zaragoza, Estado de México, México.

20 de noviembre, 2012.

Resumen

Este documento busca ser una pequeña introducción al lenguaje de programación Ada, especialmente a sus características referentes al cómputo paralelo.

1 Introducción

Ada es un lenguaje de programación de alto nivel estructurado, con tipos estáticos, y orientado a objetos que permite el desarrollo de aplicaciones de tiempo real y de gran tamaño de una manera sencilla. Más importante aún es el hecho de que tiene un gran soporte para paralelismo debido a varios mecanismos que incluye como el paso sincrono de mensajes y los objetos protegidos.

1.1 El inicio

Ada nació a finales de los años setenta como respuesta a una convocatoria realizada por el departamento de defensa de los Estados Unidos. En esta convocatoria se requería la creación de un lenguaje de programación de alto nivel para sistemas embebidos que ofreciera un buen control de tiempo real en sistemas grandes pues los lenguajes que utilizaba en aquel momento no resultaban apropiados para ello. Tras un proceso de preselección, de diecisiete propuestas recibidas quedaron cuatro a las cuales se les asignó como nombre algún color para mantener a los desarrolladores en el anonimato. Los cuatro equipos fueron:

- Intermetrics liderado por Benjamin M. Brosgol (Rojo)
- Cii Honeywell Bull liderado por Jean Ichbiah (Verde)
- SofTech liderado por John Goodenough (Azul)
- SRI International liderado por Jay Spitzen (Amarillo)

Finalmente ganó "Verde" y fue nombrado "DoD-1" en honor al departamento de defensa o "*department of defense*". Esto no le agradó a sus desarrolladores pues temían que los posibles usuarios no militares desarrollaran diferentes prejuicios debido a esta evidente relación con la milicia y optaran por no usarlo. Poco después, en 1979, Jack Cooper (del comando de equipamiento de la marina) sugirió que se le llamara "Ada". Esto en honor a Ada Lovelace, quién trabajó con Charles Babbage en lo que se considera la primera computadora jamás hecha y se convirtió en la primera programadora de la historia.
Cabe mencionar que se le pidió permiso al conde de Lytton, quien es descendiente de Ada, para usar ese nombre y que el conde mismo aceptó y mostró gran entusiasmo por ello pues en sus palabras, *las letras "Ada" están justo en medio del radar.*

1.2 ¿Por qué usar Ada?

Ada cuenta con varias ventajas u ofrece diferentes cualidades que lo convierten en una alternativa bastante interesante y atractiva para el desarrollo de software. Algunas de éstas son:

- **Seguridad:** Ada tiene algunas razones por las que se considera que es bastante seguro. Por mencionar algunas de ellas está el hecho de que sus compiladores son revisados por el gobierno de los Estados Unidos y organizaciones como la ISO y por tanto son más seguros y eficientes. También debido a que los programas de Ada son escritos en módulos independientes, es más fácil detectar algún error y corregirlo sin afectar a los demás módulos. Igualmente, gracias a la reusabilidad de los módulos en Ada se logran reducir los errores que se podrían derivar de escribir código nuevo. Además de algunas otras.
- **Desarrollo de software más fácil:** Debido a la independencia de módulos es mucho más fácil desarrollar aplicaciones con Ada pues cada programador o cada equipo puede encargarse de una sola parte del programa sin preocuparse por compatibilidad o errores que puedan surgir de la interacción entre éstas.
- **Menor costo:** Debido a la facilidad con que se lee, la posibilidad de reutilizar módulos, la escalabilidad, etcétera, Ada permite producir y dar mantenimiento a software de una manera rápida y sencilla, lo cual se traduce en un menor costo.

1.3 ¿En qué casos sería bueno usar Ada?

Ada es un lenguaje de propósito general que es especialmente bueno para desarrollar proyectos grandes de manera rápida y ágil. El hecho de que tenga una estrucutra de bloque es particularmente útil a la hora de escribir programas grandes pues permite dividir el problema en pedazos y distribuir esos pedazos entre diferentes grupos de trabajo.

2 Lo básico de Ada

Primero que nada, es importante aclarar algunas cosas que se han mencionado antes pero que no han sido explicadas. En Ada los programas son divididos en pequeños pedazos o módulos. Estos pedazos reciben el nombre de ***paquetes*** y cada uno contiene sus propios tipos de datos, procedimientos, funciones, etcétera. Uno de los procedimientos de alguno de los paquetes del programa es el que toma el lugar de lo que en otros lenguajes es "la función Main" y se encarga de declarar variables y ejecutar lo necesario para que el programa haga lo que debe de hacer, incluyendo llamadas a otros procedimientos de otros paquetes.
Quizá suene algo extraño lo dicho anteriormente, en especial lo de "sus propios tipos de datos", pero eso y algunas cosas más serán explicadas a continuación.

2.1 Tipos

Ada es un lenguaje cuyo sistema de tipos es bastante interesante. Existen los tipos predefinidos que ya tienen ciertas caracteristicas, funciones y rangos predeterminados y existe también la posibilidad de crear tus propios tipos. Independientemente de si son tipos definidos por ti o predefinidos, el sistema de tipeo de Ada se rige por cuatro reglas:

- **Tipeo fuerte:** Los datos son incompatibles entre ellos aunque sí hay maneras de convertir de un tipo al otro.
- **Tipeo estático:** Los tipos se revisan a la hora de compilar, lo cual permite detectar errores de tipos antes.

- **Abstracción:** Los tipos son representaciones del mundo real, por lo que la manera en que son representados internamente es completamente independiente y en cierto modo irrelevante, aunque sí hay maneras de definirla.
- **Equivalencia de nombres:** Solo los tipos con el mismo nombre son compatibles entre ellos.

Habiendo explicado esto, es bueno pasar a explicar un poco sobre los "tipos de tipos", aunque suene raro. Primero, los tipos predefinidos. Respecto a ellos no hay mucho que explicar, salvo qué son y como funcionan, por lo que a continuación listaré los más comunes[1].

- **Integer:** Este tipo de dato representa números en un rango que depende de la implementación del lenguaje. Además, este tipo tiene definidos dos subtipos que son los Positive (de 1 hasta Integer'Last) y los Natural (de 0 hasta Integer'Last).
- **Float:** Este tipo tiene una implementación muy débil, así que se recomienda mejor definir tu propio tipo y darle la precisión y rango deseado.
- **Duration:** Este es un tipo de punto fijo usado para medir tiempo. Representa periodos de tiempo en segundos.
- **String:** Este tipo son arreglos indefinidos y existen de tres tipos: los de un tamaño fijo, los de un tamaño que varía pero que es menor que un tope y los de tamaño variable y sin tope. Todos estos tipos tiene sus variables para los tres tipos de Character.
- **Boolean:** Este tipo es una enumeración pero solo con los valores *True* y *False* además de que tienen una semántica especial.

Ahora es momento de pasar a los tipos que se pueden definir. Respecto a ellos lo mejor será describir como se definen. Para definir un tipo se usa la siguiente sintaxis:
`type T is...` seguido por la descripción del tipo. Un ejemplo sería:

```
type Integer_1 is range 1 .. 10;
A : Integer_1 := 8;
```

Esto es posible y no marca error porque se asigna a la variable A un valor que está dentro del rango de valores del tipo `Integer_1`. Si se deseara copiar el valor de la variable A a otra variable que fuera de otro tipo, por ejemplo `Integer_2`, se marcaría un error porque los diferentes tipos son incompatibles. Además de definir tipos, se pueden definir subtipos y tipos derivados. La diferencia entre los dos es que los subtipos son compatibles entre ellos, es decir, entre subtipos mientras que los tipos derivados son compatibles con su tipo padre y heredan sus operaciones primitivas. Además, el rango de valores de los subtipos no debe estar contenido en el rango de valores del tipo del que son subtipos, mientras que en el caso de los tipos derivados si debe ser así pues las operaciones que heredan del padre suponen que el rango del tipo derivado es por lo menos una parte del rango del tipo padre.

Para definir un subtipo se usa la siguiente sintaxis:
`subtype T is...` seguido por la descripción del subtipo. Un ejemplo sería:

```
type Integer_1 is range 1 .. 10;
subtype Integer_3 is Integer_1'Base range 7 .. 11;
A : Integer_1 := 8;
B : Integer_3 := A;
```

En este caso es posible la asignación de A a B porque ambos son subtipos de la clase `Integer_1'Base` [2].

Por otro lado, para definir un tipo derivado se usa la siguiente sintaxis:
`type T2 is new T...` seguido por la descripción del tipo. Un ejemplo sería:

```
type Integer_1 is range 1 .. 10;
type Integer_2 is new Integer_1 range 2 .. 8;
A : Integer_1 := 8;
```

Ahora sí, habiendo explicado un poco de los tipos de Ada, podemos pasar a una explicación básica de la estructura de un programa.

2.2 Estructura de un programa

Primero que nada, hay que tener un programa para analizar. Ya que será un análisis sencillo, usaremos un programa sencillo. Usaremos el clásico "Hello World" escrito en Ada. El programa es:

```
with Ada.Text_IO; use Ada.Text_IO;
procedure Hello is
begin
Put_Line ("Hola mundo desde Ada!");
end Hello;
```

Primero, el comando `with` vendría a ser una especie de equivalente del `include` de C y C++. Este comando agrega el paquete `Ada.Text_IO` al programa y hace posible que se utilicen sus tipos y funciones. La palabra `procedure` indica que un procedimiento será declarado y lo que le sigue es el nombre del procedimiento. Después las palabras `begin` y `end` marcan el inicio y el final del procedimiento. Finalmente entre `begin` y `end` se escribe el cuerpo del procedimiento.

3 Lo que nos interesa, Ada concurrente

Como ya se ha mencionada algunas veces antes, Ada tiene muy buen soporte para paralelismo y concurrencia debido a la manera en que se estructuran sus programas. Para Ada, la unidad básica para la concurrencia es la tarea (*task* en inglés). Es importante mencionar que de hecho, por lo menos en cierto modo, hay dos tipos de tareas: las tareas sencillas y los tipos tarea. Las tareas simplemente son una tarea única y especial, es decir, que solo hay una de ellas. Por otro lado, un tipo tarea es una especie de plantilla para tareas y se permite tener varias tareas del mismo tipo. Las tareas tienen la capacidad de comunicarse entre ellas a través de paso de mensajes y pueden compartir variables a través de una memoria compartida. Estas características son posibles gracias a un mecanismo "de citas" (*rendezvous* en inglés) que establece un punto de sincronización entre dos tareas. Debo mencionar que este mecanismo hace que una de las tareas se suspenda hasta que la otra tarea alcance el mismo punto. Es también importante dejar claro que las tareas no son llamadas como lo son los procedimientos o las funciones, sino que comienzan a ejecutarse cuando el procedimiento principal inicia y solo se detienen para esperar los valores especificados en los puntos de entrada.

3.1 Estructura de una tarea

Las tareas y los tipos tareas comparten en cierto modo la misma estructura. Se dice esto pues ambos son declarados en dos partes que son la definición de la interfaz pública y puntos de entrada y el cuerpo de la tarea o la implementación del código que realiza en sí las funciones de la tarea. Hablando más especificamente, una tarea se declara con la siguiente estructura:

```
task T is ...;
    entry S(Variable : in type);
    entry R(Variable : out type);
end T;
```

```
task body T is
    {Aquí se declaran variables locales}
begin
    accept S(Variable : in type) do
        {Aquí se hace algo con el valor recibido, como asignarlo a la variable local}
    end S;
        {Puedes hacer algo más con el valor de la variable local}
    accept R(Variable : out type) do
        {Asigna algún valor a la variable que vas a devolver}
    end R;
end T;
```

La verdad es que la declaración de una tarea no es tan complicado ni difiere tanto de la declaración de un tipo o un procedimiento.

3.2 Estructura de un tipo tarea

La verdad es que la diferencia en sintaxis entre la tarea y el tipo tarea es muy pequeña. Basta con agregar la palabra *type* para que una tarea se convierta en un tipo tarea. Ejemplo:

```
task type T is ...;
    entry S(Variable : in type);
    entry R(Variable : out type);
end T;

task body T is
    {Aquí se declaran variables locales}
begin
accept S(Variable : in type) do
    {Aquí se hace algo con el valor recibido, como asignarlo a la variable local}
end S;
    {Puedes hacer algo más con el valor de la variable local}
accept R(Variable : out type) do
    {Asigna algún valor a la variable que vas a devolver}
end R;
end T;
```

Con la adición de esa pequeña palabra ahora nos es posible declarar diferentes instancias de la misma tarea. Por ejemplo,

```
type T_Pool is array(Positive range 1..10) of T;
My_Pool : T_Pool;
```

Cabe mencionar que la creación del tipo no genera tareas, pero la declaración de una instancia sí lo hace. En el caso anterior se generan 10 tareas al declarar `My_Pool`.

3.3 Algunas cosas más

Combinando las declaraciones de tipos, tareas, procedimientos, etcétera nos es posible crear programas que funcionen de manera paralela, pero hay algunas cosas más que es bueno conocer para hacer un mejor empleo de la concurrencia. Estas son:

- **La aceptación selectiva de llamadas a los puntos de entrada:** Permite revisar si una entrada ha sido llamada y actuar inmediatamente en caso positivo o negativo.
- **Los objetos y tipos protegidos:** Existen tres tipos operaciones posibles sobre objetos protegidos: Los procedimientos, que modifican el estado del obejto protegido y deben tener acceso exclusivo al objeto, las entradas que también modifican el estado del objeto pero a diferencia de los procedimientos, necesitan que una condición previamente definida se cumpla y las funciones que no modifican al objeto y por ende pueden ser utilizadas por diferentes tareas sobre el mismo objeto.
- **Llamadas selectivas a puntos de entrada:** Cuando se llama a una entrada puede darse el caso de que ésta se suspenda porque no se cumple una condición. En dicho caso, no se puede suspender la tarea indefinidamente por lo que se opta por usar las llamadas selectivas a puntos de entrada que permiten ya sea ofrecer una entrada alterna o una entrada cronometrada para saber cuando desechar la tarea.
- **Genéricos:** Similares a los *templates* de C++, los genéricos permiten definir unidades de compilación que contienen algoritmos independientes del tipo de dato que se use, es decir, que funcionan sin importar el tipo de dato con que se usen.

4 Conclusiones

Ada es un lenguaje bastante interesante que ha sabido mantenerse como una buena opción para los desarrolladores debido a las actualizaciones que ha tenido con el tiempo y la gran comunidad que lo respalda (incluido el departamento de defensa de los Estados Unidos).
Su estructura en bloques me parece algo rara pero relativamene sencilla de entender y su implementación de paralelismo es también muy sencilla. Claro que tiene ventajas y desventajas como todos los lenguajes, pero me parece una alternativa bastante buena, especialmente para proyectos grandes.

Notas

[1] Todos estos tipos están definidos en el paquete estándar.

[2] Al crear un tipo escalar se crea un tipo base que contiene todos los posibles valores del tipo y el tipo creado es subtipo del tipo base.

Referencias

[1] *Programming Languages Design and Implementation*
`http://www.halconia.org/escolar/sistemas_operativos/expo-1.html` Accedido el 31 de octubre del 2012.

[2] AdaCore. *AdaCore*
`http://www.adacore.com/` Accedido el 31 de octubre del 2012.

[3] Wikibooks *Wikibooks*
`http://en.wikibooks.org/wiki/Ada_Programming#Programming_in_Ada` Accedido el 31 de octubre del 2012.

[4] AdaIC. *AdaCore*
`http://archive.adaic.com/` Accedido el 31 de octubre del 2012.

[5] Ada Information Clearing House. *AdaIC.org*
`http://www.adaic.org/learn/materials/intro/part5/` Accedido el 31 de octubre del 2012.

El lenguaje de programación paralelo Chapel

Octavio Gerardo Ríos Valencia (A01160921) Erik Zamayoa Layrisse (A01165961)
Instituto Tecnológico y de Estudios Superiores de Monterrey
Campus Estado de México
Atizapán de Zaragoza, Estado de México, México.

20 de noviembre, 2012.

Resumen

Actualmente existen muchos y muy variados lenguajes de programación, de los cuales no todos tienen la capacidad de aprovechar al máximo los recursos de los equipos modernos; específicamente nos referimos a los procesadores multinúcleo. Los lenguajes capaces de utilizar estos recursos, conocidos como lenguajes de programación paralelo, suelen tener características muy convencionales y a la vez muy propias, por lo que son un tema digno de análisis. En este trabajo explicaremos un poco de la historia, generalidades, funcionalidades y ejemplos de uno de estos lenguajes de programación paralelo emergente conocido como Chapel.

Palabras clave: programación, paralelismo, programación en paralelo, lenguaje de programación, Chapel.

1 Introducción

Chapel es un lenguaje de programación paralelo emergente en el que su diseño y desarrollo está dirigido por *Cray Inc.* [1]. Chapel está siendo desarrollado como un proyecto de open-source con contribuciones de academia, industria y centros computacionales científicos.

Chapel está diseñado para mejorar la productividad de los usuarios finales mientras también sirve como un modelo portable de lenguaje de programación paralelo que pueda ser usado en clusters o bien en computadoras multinúcleo, tratando de semejar o mejorar el desempeño y portabilidad de los modelos de programación actuales como los *Message Passing Interface* (MPI).

Chapel soporta un modelo de ejecución de múltiples hilos gracias a un nivel alto de abstracción para la paralelización de la información, concurrencia y paralelismo anidado.

Es importante remarcar que el diseño de Chapel es a partir de sus propios principios, en lugar de basarse en algún lenguaje ya existente. Es un lenguaje de estructura de bloque imperativo, de fácil aprendizaje para los usuarios de C, C++, Fortran, Java, Perl, Matlab y otros lenguajes de programación populares.

El lenguaje está basado en el modelo de vista global de *High-Performance Fortran* (HPF), el cual es muy fuerte trabajando con lenguajes comunes para computación científica a un nivel de abstracción muy alto pero evita la debilidad de HPF's, la cual es que únicamente tiene como estructura de datos a los arreglos. Chapel, para corregir este problema, implementa programación multitareas y estructuras de datos arbitrarias con afinidad a nivel de objetos.

A diferencia de OpenMP que crea hilos con mucho peso y un esquema de compartir trabajo, Chapel no usa un esquema basado en hilos, sino que utiliza subcomputaciones que se pueden ejecutar de manera concurrente. Eliminando el concepto de hilo, no es necesario un manejador de los mismos, haciendo que cada módulo en el código de Chapel puede expresar su concurrencia libremente.

2 Desarrollo

2.1 Generalidades del lenguaje

Los siguientes principios fueron la guía para el diseño de Chapel:

- Programación paralela general
- Programación acorde a localidad
- Programación orientada a objetos
- Programación genérica

2.1.1 Programación paralela general

Chapel está diseñado para soportar la programación paralela general a través del uso de abstracciones del lenguaje de alto nivel. También soporta un *modelo de programación de perspectiva global* que incrementa el nivel de abstracción al expresar tanto la información como el control de flujo, comparado con los modelos de programación paralelos usados actualmente.

Perspectiva global de estructura de datos

Son arreglos y agregados de información que tienen tamaños e índices expresados globalmente, aunque su implementación esté distribuida a través de los *locales* del sistema paralelo. Un *locale* es una abstracción de unidad del acceso uniforme de memoria de cierta arquitectura. Dentro del locale, todos los hilos muestran tiempos de acceso similares a cualquier dirección de memoria.

Esta vista contrasta con la mayoría de los lenguajes paralelos, porque se acostumbra a que los usuarios particionen la información, ya sea vía manual o con ayuda de las abstracciones de los lenguajes.

Perspectiva global de control

Esto significa que el programa de un usuario comienza su ejecución en un solo hilo lógico de control y después se introduce el paralelismo a través del uso de ciertos conceptos del lenguaje. Todo el paralelismo en Chapel está implementado vía multihilos, estos hilos son creados gracias a los conceptos de alto nivel del lenguaje y manejados por el compilador y el ambiente de ejecución, en lugar de utilizar explícitamente el estilo de programación de crear hilos y unirlos, *fork/join*.

Con la programación paralela general se busca llegar a una gran variedad de arquitecturas paralelas.

2.1.2 Programación acorde a localidad

El segundo principio de Chapel consiste en permitir al usuario que opcionalmente e incrementalmente, especifique donde debería de colocarse físicamente en la máquina, la información y la computación. Tal control sobre la localidad del programa es esencialmente para lograr desempeño escalable en arquitecturas de memoria distribuida. Este modelo contrasta con el modelo *Single Program Multiple Data* (SPMD), donde este tipo de detalles son explícitamente especificados por el programador en una base de proceso por proceso.

2.1.3 Programación orientada a objetos

La programación orientada a objetos ha sido clave en incrementar la productividad entre los programadores, gracias a la encapsulación de información relacionada y funciones dentro de un solo componente de software. También soporta especialización y reúso como mecanismo para definir e implementar interfaces.

A pesar de que Chapel está basado en una orientación a objetos, no es necesario que el programador adopte un nuevo paradigma de programación para utilizar Chapel; ya que la capacidad de sus bibliotecas están implementadas utilizando objetos, por lo que el programador deberá conocer cómo utilizar la invocación de un método.

2.1.4 Programación genérica

El cuarto principio de Chapel es soporte para la programación genérica y el polimorfismo. Esta característica permite que el código sea escrito en un estilo que es genérico a través de los tipos, haciéndolo aplicable a variables de múltiples tipos, tamaños y precisiones. También permite el reúso de código, provocando que los algoritmos sean expresados sin ser explícitamente replicados por cada tipo posible.

Otra particularidad de Chapel es que soporta la iteración paralela en arreglos distribuidos, arreglos asociativos, arreglos no estructurados y en los iteradores definidos por el usuario.

Paralelismo de la información sobre arreglos distribuidos
Paralelismo de la información sobre arreglos con diferentes distribuciones
Paralelismo de la información sobre arreglos asociativos o no estructurados
Paralelismo de la información sin datos
Paralelismo de la información sobre iteradores definidos por el usuario

Con el soporte para la computación de información paralela, Chapel hace más fácil escribir esta categoría de códigos; al mismo tiempo provee las abstracciones necesarias para el programador, con las que puede escribir códigos más complicados de una manera eficiente [2].

2.2 Tareas paralelas y sincronización

Una tarea en Chapel es un contexto diferente de ejecución que corre concurrentemenre con otras tareas. Chapel provee una simple construcción, la declaración *begin*.

2.2.1 La declaración begin

La declaración begin crea una tarea para ejecutar una declaración. La sintaxis para la declaración begin es la siguiente:

begin-statement:
begin *statement*

El control continúa concurrentemente con la declaración siguiente de la declaración begin.

begin writeln ("output from spawned task");
writeln ("output from main task");

La salida en la terminal es no determinística.

2.2.2 Variables de sincronización

Las variables de sincronización tienen un estado lógico asociado con su valor. El estado puede ser *full* o *empty*. En modo lectura de una variable de sincronización no puede proceder hasta que el estado de la variable sea *full* y viceversa en modo escritura no se puede proceder hasta que el estado de la variable sea *empty*.

Chapel tiene dos tipos de variables de sincronización: *sync* y *single*. Ambos tipos se comportan de manera similar, excepto que la variable single solo puede ser escrita una sola vez. Esto quiere decir que cuando una

variable *sync* es leída, cambia su estado a *empty*, mientras que si una variable de tipo *single* es leída, ésta no cambia de estado. Cuando cualquiera es escrita, cambian su estado a *full*.

Cuando una tarea intenta leer o escribir una variable de sincronización que no está en un estado correcto, la tarea es suspendida. Cuando hay más de una tarea bloqueada en espera por la transición del estado, una es elegida no determinísticamente, mientras que las demás continúan en espera.

Ejemplo:

```
var count$: sync int = 0;
begin count$ = count$ + 1;
begin count$ = count$ + 1;
begin count$ = count$ + 1;
```

2.2.3 La declaración cobegin

La declaración cobegin es usada para introducir concurrencia en un bloque. La sintaxis para la declaración cobegin es la siguiente:

cobegin-statement:
cobegin *block-statement*

Es importante mencionar que una tarea es creada por cada declaración en el bloque.

Ejemplo:

```
cobegin{
    stmt1();
    stmt2();
    stmt3();
}
```

Lo equivalente a esto sería escribir una declaración begin por cada statement.

2.2.4 El ciclo coforall

El ciclo coforall es una variante de la declaracaión cobegin en forma de ciclo. La sintaxis del ciclo coforall es:

coforall-statement:
coforall *index-var-declaration* **in** *iteratable-expression* **do** *statement*
coforall *index-var-declaration* **in** *iteratable-expression block-statement*
coforall *iteratable-expression* **do** *statement*
coforall *iteratable-expression block-statement*

Ejemplo:

```
coforall i in iterator (){
    body();
}
```

2.2.5 La declaración sync

La declaración *sync* actúa como una unión de todos los begin dinámicos de una declaración. Su sintaxis es la siguiente:

sync-statement:
sync *statement*
sync *block-statement*

Ejemplo:

sync for i **in** 1. .n **do begin** work();

El ciclo for está dentro de la declaración sync, por lo que todas las tareas creadas en cada iteración del ciclo deberán completarse antes de pasar a lo que sigue de la declaración.

2.2.6 La declaración serial

La declaración *serial* puede ser utilizada para dinámicamente deshabilitar el paralelismo. La sintaxis es:

serial-statement:
serial *expression* **do** *statement*
serial *expression block-statement*

La expresión es evaluada a un tipo booleano, si la evaluación regresa verdadero, cualquier código que resulte en nuevas tareas es evaluado sin crearlas; es decir la ejecución es serializada.

Ejemplo:

```
proc f(i) {
    serial i<13 {
        cobegin {
            work(i);
            work(i);
        }
    }
}
for i in lo. . hi{
   f(i);
}
```

La declaración serial en f() inhabilita la ejecución concurrente de work(), si la variable i es menor a 13.

2.2.7 Declaraciones atómicas

La declaración *atomic* es usada para especificar que una declaración debe parecer ser ejecutada atómicamente, desde la perpectiva de otras tareas. Particularmente ninguna tarea verá memoria en un estado que refleje el hecho de que una declaración atómica ha comenzado a ejecturase y que no ha terminado.

Esta definición de la declaración atómica provee una notación de *atomicidad fuerte* debido a que la acción aparecerá atómica a cualquier otra tarea desde cualquier punto en su ejecución. Por razones de desempeño, podría ser más práctico una *atomicidad débil* en el que el estado de atomicidad sea solo garantizado con respecto a otras declaraciones atómicas. También se busca utilizar calificadores del tipo atómico como medio para marcar la información que debe ser accedida atómicamente dentro o fuera de una sección atómica.

La sintaxis es:

atomic-statement:
atomic *statement*

Ejemplo:

```
proc Node.insertAfter (newNode: Node) {
    atomic {
        newNode.prev =this;
        newNode.next =this.next;
        if this.next then this.next.prev = newNode;
        this.next = newNode;
    }
}
```

El ejemplo ilustra el uso de la declaración atomic para realizar una inserción en una lista doblemente encadenada. Esto previene que otras tareas vean la lista en un estado parcialmente actualizado donde no es consistente aún.

2.3 Paralelismo de la información

Chapel provee dos construcciones paralelas de la información explícitas, la declaración forall y la expresión forall; así como muchos lenguajes que soportan la paralelización de la información implícitamente, como: asignación de todo el arreglo, reducciones y scans.

2.3.1 La declaración forall

La declaración forall es una variante concurrente de la declaración for. Su sintaxis es la siguiente:

forall-statement:
forall *index-var-declaration* **in** *iteratable-expression* **do** *statement*
forall *index-var-declaration* **in** *iteratable-expression block-statement*
forall *iteratable-expression* **do** *statement*
forall *iteratable-expression block-statement*
[*index-var-declaration* **in** *iterable-expression*] *statement*
[*iterable-expression*] *statement*

La declaración forall evalúa el cuerpo del ciclo una vez por cada elemento dado por la expresión iterable. Cada instancia del cuerpo del ciclo forall puede ser ejecutado concurrentemente con otros, pero no está garantizado. Particularmente el ciclo debe ser serializado.

Esto se diferencia de la semántica del ciclo coforall, donde se garantiza que cada iteración corra en una tarea diferente. En práctica el número de tareas que deben ser usadas para evaluar un ciclo forall es determinado por los objetos o iteraciones que están dirigiendo la ejecución del ciclo, así como el mapeo de iteraciones de las tareas.

El control continúa con la declaración siguiente del ciclo forall solo después de que cada iteración haya sido totalmente evaluada. En este punto todos los accesos de información dentro del cuerpo del ciclo forall serán grantizados su terminación.

Ejemplo:

```
forall i in 1. .N do
   a(i) =b(i);
```

En este código el usuario ha establecido que la asignación clave puede ejecutarse concurrentemente. Este ciclo podría ejecutarse serialmente en una sola tarea o usando una tarea diferente por cada iteración o usando un número de tareas donde cada tarea ejecuta un número de iteraciones.

2.3.2 La expresión forall

La expresión forall es una variante concurrente de la expresión convencional for y su sintaxis es la siguiente:

forall-expression:
forall *index-var-declaration* **in** *iteratable-expression* **do** *expression*
forall *iteratable-expression* **do** *expression*
[*index-var-declaration* **in** *iterable-expression*] *expression*
[*iterable-expression*] *expression*

La expresión forall sigue la misma semántica de la declaración forall.

2.3.3 Configuración de constantes para la paralelización de información por defecto

La siguientes constantes de configuración son utilizadas para controlar el grado del paralelismo de la información en rangos, y arreglos por defecto:

Config Const	**Type**	**Default**
dataParTasksPerLocale	int	Number of cores per locale
dataParIgnoreRunningTasks	bool	true
dataParMinGranularity	int	1

La configuración de *dataParTasksPerLocale* especifica el número de tareas a utilizar cuando se ejecuta un ciclo forall en un rango, dominio o arreglo. Si se utiliza el valor por defecto, se usa un cero.

La configuración de *dataParIgnoreRunningRasks*, cuando es verdadero, no tiene efecto en el número de tareas a utilizar cuando se ejecuta un ciclo forall. Cuando es falso, el número de tareas por locale es disminuido por el número de tareas que actualmente estan corriendo en el locale, con un valor mínimo de uno.

La configuración de *dataParMinGranularity* especifica el número mínimo de iteraciones por tarea creada. El número de tareas es disminuido, por lo que el número de iteraciones por tarea nunca es menos que el valor especificado [3].

3 Conclusiones

Chapel podría paracer como cualquier otro lenguaje de programación, pues comparte muchas características similares a los que ya hemos estudiado. Soporta programación orientada a objetos como C++, Java, etc, tiene manejo de *reduce* como Erlang o Clojure; pero el verdadero potencial de Chapel es que su arquitectura y diseño lo vuelven un lenguaje de programación fácil de utilizar, cuenta con distintas declaraciones para paralelizar y evita el uso de manejadores de hilos, lo cual lo hace sumamente práctico.

También podemos percibir que Chapel se enfoca en la eficiencia, por la forma en que maneja sus multitareas y provee herramientas poderosas para el programador, brindándole la oportunidad de desarrollar con un poco más de libertad que con otros lenguajes; un ejemplo de esto es que permite que el programador sea libre de utilizar y manejar sus propios iteradores paralelos y que utilice la programación acorde a la localidad, donde especificará en donde deberá ir tanto la información como el poder de cómputo.

4 Agradecimientos

Queremos agradecer especialmente a **Sasha Alexandra**, una amiga que nos sugirió un editor de LaTeX mucho más amigable, *TeXstudio* y nos resolvió varias dudas en la codificación de nuestro artículo, haciendo de este proyecto una tarea más sencilla.

Referencias

[1] Cray Inc. *Cray The Supercomputer Company*
http://www.cray.com/Home.aspx Accedido el 28 de octubre del 2012.

[2] Deitz, S, Chamberlain, B, Choi, S, et all. *Five Powerful Chapel Idioms.*
http://chapel.cray.com/publications/cug10.pdf Accedido el 29 de octubre del 2012.

[3] Cray Inc. *Chapel Language Specification Version 0.92* Cray Inc, 18 de Octubre de 2012, http://chapel.cray.com/spec/spec-0.92.pdf Accedido el 28 de octubre del 2012.

Cilk para un C más facil

Enrique Fabián García Araico (A00965173) Esteban Pérez Mejía (A01163982)
Instituto Tecnológico y de Estudios Superiores de Monterrey
Campus Estado de México
Atizapán de Zaragoza, Estado de México, México.

31 de octubre, 2012.

Resumen

Este documento petende mostrar cómo generar paralelismo en C, de una manera que solo implica seis palabras clave. Todo esto de la mano de Cilk.

1 El lenguaje Cilk

Cilk es un lenguaje algorítmico basado en múltiples threads. La idea de Cilk es que un programador debe concentrarse en estructurar su programa en forma paralela sin tenerse que preocupar por como será la corrida en el sistema para mejorar su eficiencia en la plataforma. La corrida de un programa Cilk se encarga de detalles como el balanceo de carga y comunicación entre los procesadores. Cilk básicamente se asegura de que se asignen las cargas de trabajo de forma eficiente y con un desempeño predecible.

2 Usando Cilk

El lenguaje Cilk es bastante sencillo si ya sabes C. Consiste en el lenguaje C con seis palabras claves para ocuparse del paralelismo y la sincronización. Un programa en Cilk tiene la misma semántica que un programa en C si se eliminan las palabras claves de Cilk. Cuando se corre un programa en un procesador y sin estas palabras claves el programa es llamado "serial eleison" o un "C eleison" que básicamente significa que el programa en Cilk tiene el mismo desempeño que la versión de C.

Un ejemplo para corrobborar esto es el siguiente:

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

int fib (int n)
{
   if (n<2) return (n);
   else
   {
      int x, y;

      x = fib (n-1);
      y = fib (n-2);

      return (x+y);
   }
}

int main (int argc, char *argv[])
{
   int n, result;

   n = atoi(argv[1]);
   result = fib (n);

   printf ("Result: %d\n", result);
   return 0;
}
```

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

int fib (int n)
{
   if (n<2) return n;
   else
   {
      int x, y;

      x = spawn fib (n-1);
      y = spawn fib (n-2);

      sync;

      return (x+y);
      }
}

cilk int main (int argc, char *argv[])
{
   int n, result;

   n = atoi(argv[1]);
   result = spawn fib (n);

   sync;

   printf ("Result: %d\n", result);
   return 0;
}
```

Como se puede ver en el código anterior, los programas muestran el enésimo numero de Fibonacci. El programa de la izquierda esta hecho en C y lo realiza de una forma recursiva, mientras que el de la izquierda está en lenguaje Cilk y lo realiza de forma paralela. Se puede ver como los dos programas se ven casi idénticos a excepción de que el de Cilk tiene tres palabras clave nuevas: `cilk`, `spawn`, `sync`. Si se quitaran estas palabras se convertiría en un programa en C que correría en un procesador, dígase un "C eleison".

Las palabras claves que utiliza Cilk es lo que lo diferencia de un programa de C y lo que permite usar paralelismo. La palabra clave `cilk` identifica una función paralela en C. Una función con la palabra `cilk` puede llamar subprocesos en forma paralela para al final sincronizarlos cuando se completen. Solo se debe poner la palabra `cilk` en una función que deseas que sea paralela y poner todo lo demás como cualquier función de C. El uso de la palabra `cilk` en una función únicamente la identifica como una creadora de subprocesos pero no la hace paralela en sí. Para hacerlo de ese modo, se utiliza otra palabra clave que es `spawn`. Básicamente, `spawn` es una forma paralela de hacer un llamado a la función, lo que genera un hijo con ese método para ejecutar.

2.1 Diferencia entre C y Cilk

La diferencia de C y Cilk en la creación de subprocesos, es que en C el procedimiento padre debe esperar a la terminación del hijo para continuar con su ejecución, mientras que en Cilk, el padre puede continuar su ejecución de forma paralela al hijo. Esto provoca que el padre sea capaz de llamar a mas hijos a realizar subprocesos lo que da un alto grado de paralelismo. Y como se menciona al principio, no hay que preocuparse por balanceo de carga entre los procesadores, ya que Cilk asignara la carga según su algoritmo lo vea mas eficiente.

En esta imagen se muestra como un padre genera hijos y los hijos generan más hijos y esto lo realiza de forma paralela. El padre no esperara a que los hijos terminen para seguir con su ejecución y continuara generando hijos.

Esto puede llegar a generar un problema, ya que si todo va en forma paralela, no se pueden regresar datos de los hijos en forma ordenada lo que podría ocasionar una condición de carrera. Para evitar las condiciones de carrera se usa la palabra clave sync, la cual se encargara de esperar a que todos los hijos acaben su ejecución para usar los datos que regresan. Cuando se usa sync, se genera una barrera local que esperara únicamente a los procesos que se hayan creado desde la función cilk. Esto hace que se espere únicamente a los hijos y no a todos los procedimientos que se estén ejecutando. Cuando los hijos hayan terminado, se continuara con la ejecución normal del procedimiento. Como una ayuda que ofrece cilk, siempre habrá un sync implícito antes de cada return lo que provoca que siempre acaben los hijos antes que el padre para continuar de forma ordenada su ejecución.

Ejemplo

```
cilk int foo (void)
{
   int x = 0, y;

   spawn bar(&x);
   y = x + 1;
   sync;
   return (y);
}

cilk void bar (int *px)
{
   printf("%d", *px +1);
   return;
}
```

El sync implícito no asegura que no haya errores de cálculo por condiciones de carrera. Un ejemplo de este tipo de situación se muestra a continuación.

a)

```
cilk int foo (void)
{
   int x = 0;

   spawn bar(&x);
   sync;
   x = x + 1;
   return (y);
}

cilk void bar (int *px)
{
   p*px = *px + 1;
   return;
}
```

Caso que no presenta condición de carrera, ya que el sync se hace antes de utilizar la variable x en el cálculo `x = x + 1`.

b)

```
cilk int foo (void)
{
   int x = 0;

   spawn bar(&x);
   x = x + 1;
   return (y);
}

cilk void bar (int *px)
{
   p*px = *px + 1;
   return;
}
```

Caso que presenta condición de carrera, ya que el sync se hace implícito antes del return, esto hace que la acción `x = x + 1` se haga de manera no determinística ya que no se espera a obtener el resultado de bar.

2.2 Estructura de Cilk

Como ya dijimos, un programa de Cilk está basado en un programa de C. Además de esto se tienen definiciones y declaraciones de tipo Cilk. El programa de Cilk, al igual que uno de C, tiene un método main que toma argumentos de la línea de comandos y regresa un entero. Las funciones de cilk pueden usar funciones de C, pero en una función de C no se pueden usar funciones de tipo Cilk. Para esto se requiere especificar que la función es tipo Cilk con la palabra clave `cilk` y de ahí se puede usar todo de Cilk y de C.

Las palabras clave que se utilizan son las mismas que C y además unas extras que se definen en Cilk. Estas palabras son: `cilk, spawn, sync, inlet, abort, shared, private` y `SYNCHED`. Para definer metodos en Cilk se realiza del mismo modo que en C, salvo con la excepción de que se pone la palabra `cilk`. Esto define un tipo Cilk y permite usar las palabras clave de Cilk en el método. Cabe remarcar que si se usa un método tipo Cilk, se deben llamar procedimientos como tipo Cilk con `spawn` ya que no se permite usar una invocación ordinaria como la de C.

La palabra clave `spawn` creará un subproceso o hilo que se encargara de la carga de trabajo en forma paralela. Sin embargo tiene ciertas normas que hay que seguir para poderla usar. Las funciones llamadas con un spawn pueden regresar o no algo, pero si regresan algo, se tiene que asignar a una variable del mismo tipo de regresó. Por ejemplo si una función Cilk invocada con `spawn` regresa un float, una variable tipo float tiene que ser la que recibe el resultado. No se puede hacer conversión de tipos como de un float a un int. Dígase que si intentas recibir el resultado del ejemplo anterior en un int, te marcara un error ya que forzosamente debe residir en una variable del mismo tipo.

2.3 Más acerca de spawn

Los operadores en un `spawn` son bastante sencillos, pero se debe considerar lo siguiente: la sintaxis de un `spawn` es un statement, no una expresión. Debido a esto no se puede poner algo como:

```
a = spawn foo() + spawn bar();
```

Esto, debido a que el spawn no es una expresión. Por ello no se pueden usar operadores entre spawns. Si se quiere realizar operaciones entre los regresos de cada método se deberán usar los siguientes operadores:

```
=  *=  /=  %=  +=  -=  <<=  >>=  &=  ^=  |=
```

Solamente se podrán usar esos operadores cuando se usan spawns. En el caso del regreso de los spawns, son idénticos a C. Pones un return y el valor que quieres devolverle al padre.

2.4 Más acerca de sync

La palabra clave `sync` básicamente, es un statement que pondrás en el método para poder sincronizar el regreso de todos los hijos. Simplemente es una instrucción que esperara a la ejecución de todos los hijos para que la memoria compartida sea coherente y se eviten condiciones de carrera. Este se puede poner en cualquier parte del método para controlar donde se debe esperar el regreso y se puede poner más de una vez para saber a que hijos esperar y a cuales no.

2.5 Inlets

Como ya vimos, los spawns o hijos no te permiten hacer expresiones debido a que son statements. Por ello, si la función regresa algo, se tiene que almacenar en algún punto para después usarlo. Si se quiere usar directamente el resultado que regresa un método se puede usar un inlet. El inlet es como una función de C que recibirá lo que regrese el argumento que se mande dentro del inlet. Un inlet al ser una función dentro de otra, podrá usar las variables del padre ya que tiene el alcance (scope) para usarlas.

Así mismo puede haber inlets implícitos. Es básicamente una trampa ya que los explicamos anteriormente pero no los definimos como inlets, sino como parte de la sintaxis del spawn. Cuando un spawn usa alguno de sus operadores a excepción del '=', se define un inlet implícito que permite hacer la operación del spawn. El uso de inlets permite que los resultados de un hijo puedan usarse en el padre para alcanzar la solución. Eso sería en teoría lo que es un inlet, pero hay que tener en cuenta ciertas consideraciones al usarlo.

La palabra clave `inlet` es una un poco más complicada. Inicialmente se refiere a un pedazo de código que se ejecutara cuando alguno de los hijos regresa. Éste tiene que ser definido en la parte de declaración del método. Lo importante de un `inlet`, es que se ejecutara cuando el hijo regresa y lo hará de forma atómica, separada de los procedimientos tipo Cilk y de los demás inlets. Para poder hacer un `inlet` se tiene que usar la palabra clave `inlet`, el tipo del `inlet`, el nombre del mismo, los argumentos del `inlet` y un cuerpo que consiste en statements de C. Dentro del cuerpo se pueden usar la palabra clave `abort` o `SYNCHED` pero ninguna otra de parte de Cilk.

Los inlets ejecutan su cuerpo cuando el procedimiento Cilk ha terminado y puede usar los argumentos que se le mandan. Cuando se ejecuten los hijos, estos harán su trabajo y cuando terminen enviarán su valor al `inlet`, el cual podrá modificarlo de manera atómica para usarlo después. En el caso de que el `inlet` tenga un tipo de regreso, este se deberá asignar a otro del mismo tipo (al igual que con `spawn`). Esto sucede igual con los argumentos que le pases al `inlet` y lo que regrese.

2.6 abort

Un caso especial a considerar en el paralelismo, es que se pueden usar multiples funciones para hallar una sola solución. Esto en algunos casos implica que varias posibles soluciones son probadas en paralelo, sin embargo hay situaciones en las que solo nos interesa una solución y no todas las posibles, por lo que preferimos quedarnos con la primera que aparezca.

Uno de los problemas con esta situación es que muchas veces, cada ramificación que el algoritmo genera para paralelizar la búsqueda de la solución, sigue trabajando aún después de que se ha encontrado esta. Para este típo de situaciones se puede utilizar la palabra `abort`. Esta palabra clave es algo obvia. Aborta la ejecución de algún hijo. Esto es para alivianar carga de trabajo y procedimientos que ya no hagan nada.

Básicamente se usa para interrumpir prematuramente la ejecución de un hijo que ya hizo su trabajo o que

esta haciendo trabajo innecesario. Obviamente todo el trabajo que haya realizado el hijo hasta el momento será descartado y puede o no pasar al padre dependiendo de su regreso. La variable `SYNCHED` permite a un procedimiento determinar el progreso de los hijos que creó. Es una variable que tendrá un 1 si sus hijos han terminado con operaciones en memoria y 0 si no es así. Esta es una variable read-only que solo puede ser usada en un `inlet` o un método tipo cilk.

2.7 compilación de un programa Cilk

Para compilar un programa Cilk se usa una distribución que solo es una versión especial del compilador gcc. Cilk 5.4.6 automaticamente instala el comando `cilkc` que actúa de forma idéntica a gcc. La diferencia más grande de este compilador es que además te ofrece diversas opciones para que se muestre información adicional con la corrida del programa. Por ejemplo, si cuando compilas pones la bandera `-cilk-profile`, te mostrará cuanto tiempo tardó cada procesador, cuantos threads se generaron, cuanta memoria se usó, etc. Esta información te será útil para ver cómo es tu paralelismo y la carga de trabajo que mandaste.

La compilación de `cilk` de hecho es un poco más compleja que la de un programa en C. Primero el archivo .cilk y el header se tienen que agregar a otro archivo .cilkI. Despues el archivo .cilkI pasa por el preprocesador de C, lo que produce un archivo .cilki. Ahora el archivo .cilki es procesado por cilk2c, que es un traductor encargado de pasar de cilk a C, y genera un archivo .cilkc. El archivo .cilkc pasa de nuevo por el preprocesador de C y genera un archivo con extensión .i y por ultimo gcc se encarga de archivos con ese tipo de extensión. El compilador de cilk admite muchos argumentos de gcc, pero no todos. En el manual de cilk se describen todos los argumentos que se pueden usar de parte de gcc.

2.8 Memoria en cilk

El almacenamiento de memoria en Cilk es bastante parecida a la de C. Se trabaja con 2 tipos de memoria: Stack y un heap. La memoria Stack se asigna por el compilador y se libera cuando el método termina. La memoria heap se asigna con un `Malloc()` y se libera con un `Free()`. La memoria heap es como la de C. Cilk usa un tipo de Stack que se denomina Cactus Stack. Es bastante parecida a una Stack cualquiera, la única diferencia es que cada padre tendrá un stack de los hijos que ha invocado, pero un hijo no podrá ver a su padre. Ésto produce que en forma paralela se generen vistas del stack que contendrán la información de los hijos. Ésta memoria básicamente es una como la de C, con la diferencia de que al ser paralelas, se generaran varias vistas del Stack y cada una con su historia de invocaciones y variables.

2.9 Memoria compartida en cilk

La memoria compartida en Cilk también se puede usar en C, pero al igual que en C y en otros lenguajes, esto puede producir inconsistencias. Para compartir datos puedes usar un apuntador o variables goblales. Pero esto puede provocar condiciones de carrera en esas variables. Lo más prudente en este lenguaje, es hacer lo que harias en cualquier otro lenguaje: "evita escribir variables compartidas". El modelo de memoria compartida en cilk se debe usar con precaucion. La consistencia de la memoria es muy importante por lo que Cilk pone también primitivas que hacen que cada instrucción se ejecute de manera atómica. Una de estas primitivas es el `cilk_fence()` que hace que se cumpla primero una instrucción antes de pasar a la siguiente.

2.10 Locks

Cilk también tiene `locks` para excluir partes importantes del código. Para usar estos locks, solamente se tiene que crear un `lock` tipo `cilk_lockvar`, inicializarlo y bloquear lo que se gusta. Trabajan exactamente igual que un `locks` cualquiera. Para crearlo es solo como crear una variable tipo `cilk_lockvar`, para inicializarlo se usa `cilk_lock_init` que recibe como parámetro un `lock` de tipo `cilk_lockvar`, y para bloquear y liberar

código se utiliza `cilk_lock` y `cilk_unlock`. Estos últimos reciben de parámetro el mismo `lock` que ya tiene que estar inicializado.

3 Conclusión

En este artículo podemos concluir que Cilk es una implementación muy natural de paralelismo para C y C++, ya que, al incluir pocas instrucciones es facil de aprender y dificil de cometer errores. El hecho de que sea compatible con C y C++ lo hacen ideal para una gran cantidad de proyectos.

Referencias

[1] Massachusetts Institute of Technology. *Cilk 5.4.6 Reference Manual*
http://supertech.csail.mit.edu/cilk/ Accedido el 21 de octubre del 2012.

[2] KNOX College *Cilk Tutorial*
http://faculty.knox.edu/dbunde/teaching/cilk/ Accedido el 22 de octubre del 2012.

Concurrencia en Curry

Luis Reyes (A01160463)
Instituto Tecnológico y de Estudios Superiores de Monterrey
Campus Estado de México
Atizapán de Zaragoza, Estado de México, México.

31 de octubre, 2012.

Resumen

Curry es un lenguaje de programación universal y multi-paradigmático que conjunta la programación funcional, la programación lógica y programación de restricciones. La forma en que implementa la concurrencia es muy sencila para el programador y lo hace por medio de restricciones.

1 Introducción

Los lenguajes de programación declarativos tienen la característica de que al programar se les expresan las propiedades de los problemas y de las soluciones en general, en contraste con los lenguajes imperativos. Los principales paradigmas presentados en el artículo [3] son:

- Lenguajes Funcionales: Se basan en el cálculo lambda, no maneja datos mutables. Los programas son un conjunto de funciones definidas en ecuaciones que se utilizan para evaluar expresiones de izquierda a derecha y, debido a la falta de construcciones naturales como las iteraciones, se utiliza la recursión para la repetición de instrucciones.
- Lenguajes Lógicos: Se basan en un subconjunto de la lógica de predicados para hacer relaciones entre elementos, de esa forma se garantiza un modelo de ejecución efectiva de primer orden.
- Lenguajes de Restricciones: Se basan en el uso de restricciones para relacionar variables. Una vez definido el conjunto de restricciones se encuentra la solución que satisface dicho conjunto sin especificar los pasos a seguir para obtener la solución.

Curry es un lenguaje de programación universal, multi-paradigmático, fuertemente tipado, con inferencia de tipos y tipado estático que tiene como objetivo principal conjuntar los paradigmas más importantes de programación declarativa: la programación funcional, la programación lógica y programación de restricciones [6]. Además, abarca los principios operativos más importantes desarrollados en el área de lenguajes lógicos-funcionales: *residuation* y *narrowing*.

Curry combina una serie de características de la programación funcional (expresiones anidadas, funciones de orden superior, *lazy evaluation*), de la programación lógica (variables lógicas, estructuras parciales de datos, *built-in search*), y la programación concurrente (evaluación concurrente de las expresiones con la sincronización en variables lógicas). El desarrollo de Curry es una iniciativa internacional que surgió la decada pasada cuyo objetivo es proporcionar una plataforma común para la investigación, la enseñanza y la aplicación de lenguajes lógicos-funcionales. Su principal diseñador es Michael Hanus.

En este artículo se dará una visión general del lenguaje y las características principales para implementar concurrencia.

2 Desarrollo

2.1 Visión general de Curry

Curry tiene una sintaxis muy parecida a la del lenguaje funcional Haskell, ya que está basado en éste. Los nombres de las funciones y variables empiezan con minúscula y los constructores de datos así como los tipos empiezan con mayúsculas. El uso de funciones se denota con el nombre de la función seguido de sus argumentos a excepción de los operadores infijos que pueden ser escritos de forma natural para mantener una notación matemática estándar; a esta notación se le conoce como currificada. La característica principal que separa a Curry de un lenguaje funcional puro es la posibilidad de incluir variables *free*, que son características de los lenguajes lógicos.

Las funciones en Curry se definen por medio de expresiones, pero éstas reciben un nombre y usualmente utilizan parámetros para que sean utilizadas repetidas veces en el programa cambiando sólo los argumentos, evitando así código repetido. Una expresión puede ser un *atom*[1] o la aplicación de una expresión a otra expresión.

Hay funciones sin parámetros:

```
doce = 6 + 6
```

Y con parámetros:

```
potencia2  x = x * x
```

Una vez que son definidas las funciones para ser evaluadas sólo se necesita escribirlas en un archivo con extensión .curry y cargarlo desde la línea de comando del ambiente `:load test`, en este paso se utiliza la implementación de *PACKS*[2] [4] y el archivo test.curry.

```
test> potencia2 doce
Result: 144
More solutions [Y(es)/n(o)/a(ll)]?
```

Curry cuenta con especificación de tipos, es decir se puede especificar los tipos de entrada y salida. También soporta el estilo de *pattern-oriented* así como el uso de variables anónimas representadas con el carácter "_".

Curry permite la definición de funciones de varias reglas y es capaz de buscar varias soluciones. Se puede combinar ambas características para definir funciones que producen más de un resultado para una entrada específica, esta característica es heredada del paradigma lógico. Tales funciones se llaman funciones no deterministas o *set-valued*. Por ello, el último renglón del código anterior está en espera de una entrada para saber qué acción ejecutar entre buscar otra solución, terminar la evaluación o encontrar todas las posibles soluciones; pero en este caso no existe otra solución.

Una función que sí tiene soluciones múltiples es la siguiente:

```
escoge x y = x
escoge x y = y

test> escoge 6 9
Result: 6
More solutions? [Y(es)/n(o)/a(ll)] y
Result: 9
More solutions? [Y(es)/n(o)/a(ll)] y
No more solutions.
```

Al ser evaluada, se pueden obtener todos sus valores escogiendo la opción `y`. Para una referencia más específica se puede consultar el reporte del lenguaje disponible en [2] y el tutorial básico en [5].

2.2 Características concurrentes

Curry ofrece una forma muy sencilla y transparente para incorporar concurrencia en sus programas. Esto lo logra al momento de ejecutar restricciones con ayuda de variables *free*. Este tipo de variables se encuentran sin instanciar o sin relacionar. El objetivo principal al tener restricciones y variables *free* es asignarle valores a las variables hasta que la expresión sea reducible, esto significa que la expresión llegue a un caso terminal y se satisfaga la restricción.

2.2.1 Restricciones

En Curry existe el tipo *Boolean* como en muchos lenguajes para realizar álgebra booleana y evaluar condiciones, pero para poder evaluar restricciones se debe de utilizar un tipo y los operadores especiales siguientes:

Tipo:

Tipos	Declaración	Ejemplo
Success	Success	success, failed

El tipo *Success* no tiene valores literales y su objetivo es denotar el resultado de una restricción, usualmente se utiliza para comprobar satisfactibilidad.

Operadores:

Descripción	Identificador
Igualdad de restricción	=:=
Conjunción paralela	&
Restricción de expresión	&>

La igualdad de restricción aplica en expresiones como u y v, es decir, `u =:= v`, tiene éxito si y sólo si, u y v se puede evaluar al mismo valor de lo contrario falla y no se devuelve ningún valor.

La conjunción paralelo se aplica a expresiones u y v , es decir, `u & v`, u y v se evalúan al mismo tiempo. Si ambas son exitosas la evaluación también lo es, de lo contrario falla.

La restricción de expresión es aplicada a una restricción c y una expresión, es decir, `c &> e`, se evalúa c primero y si esta evaluación tiene éxito, inmediatamente se evalúa e, de lo contrario se produce un error.

Éste es un ejemplo utilizando restricciones, `data` se utiliza para definir tipos definidos por el usuario.

```
data Persona = LukeS | CadeS| Leia0 | DarkV
padre :: Persona -> Persona
padre LukeS = DarkV
padre CadeS = LukeS
padre Leia0 = DarkV
```

Al procesar un hijo de `DarkV`, la variable x tiene que ser definida como *free* y es inicializada a dos posibles soluciones.

```
test> padre x =:= DarkV where x free
Free variables in goal: x
Result: success
Bindings:
x=LukeS
More solutions? [Y(es)/n(o)/a(ll)] a
Result: success
Bindings:
x=LeiaO
No more solutions.
```

De forma similar, podemos obtener de quién es abuelo `DarkV` como se muestra a continuación:

```
test> padre (padre x) =:= DarkV where x free
Free variables in goal: x
Result: success
Bindings:
x=CadeS
More solutions? [Y(es)/n(o)/a(ll)] y
No more solutions.
```

2.2.2 Evaluación

Una de las características principales de Curry es la evaluación de expresiones que tienen variables tipo *free*. Hay dos técnicas para realizar la evaluación de las expresiones que contienen variables *free*: *residuation* y *narrowing*.

Por ejemplo, supongamos que se tiene una expresión a evaluar e y una variable v contenida en e. Además, supongamos que e no puede ser evaluada porque el valor de v es desconocido, la *residuation* suspende la evaluación por lo que no genera un resultado. A este tipo de operaciones se les conoce como rígidas y son principalmente operaciones aritméticas:

```
Prelude> x == 40 + 2 where x free
Free variables in goal: x
*** Goal suspended!
Bindings:
x=_6299
*** Warning: there are suspended constraints (for details: ":set +suspend")
```

Ahora, con la misma suposición se puede utilizar la técnica de *narrowing*. En contraste con *residuation* debido a que e no puede ser evaluada porque se desconoce el valor de v, al utilizar *narrowing* se infiere un valor para v hasta que encuentra la solución en un conjunto especifico. A este tipo de operaciones se les conoce como flexibles y se utiliza el operador de igualdad de restricción:

```
Prelude> x =:= 40 + 2 where x free
Free variables in goal: x
Result: success
Bindings:
x=42
More solutions? [Y(es)/n(o)/a(ll)] a
No more solutions.
```

2.2.3 Ejemplos

Para poder ejemplificar la concurrencia en acción se tiene este pequeño programa:

```
digito :: Int -> Success
digito 0 = success
digito 1 = success
digito 2 = success
digito 3 = success
digito 4 = success
digito 5 = success
digito 6 = success
digito 7 = success
digito 8 = success
digito 9 = success
```

Se define la función dígito que recibe un entero y regresa un *Success* para representar el dominio del problema y se introducen los dígitos del 0-9.

Después se ejecuta el código:

```
test> x+x=:=y & x*x=:=y & digito x & digito y where x, y free
Free variables in goal: x, y
Result: success
Bindings:
x=0
y=0
More solutions? [Y(es)/n(o)/a(ll)] a
Result: success
Bindings:
x=2
y=4
No more solutions.
```

Como se mencionó anteriormente, el operador & ejecuta de forma concurrente las restricciones `x+x=:=y` y `x*x=:=y` resultando en dos posibles soluciones al problema. Si se cambia el regreso de los dígitos que son parte de las soluciones a *failed*:

```
digito :: Int -> Success
digito 0 = failed
digito 1 = success
digito 2 = failed
digito 3 = success
digito 4 = failed
digito 5 = success
digito 6 = success
digito 7 = success
digito 8 = success
digito 9 = success
```

Ahora ya no existe solución alguna:

```
test> x+x=:=y & x*x=:=y & digito x & digito y where x, y free
Free variables in goal: x, y
No more solutions.
```

Otro ejemplo es el típico problema criptográfico "$send + more = money$" donde a cada letra s, e, n, d, m, o, r, y se le asigna un dígito del 0 al 9 que cumpla con $send + more = money$".

Como se explica en el libro [1], la forma más sencilla de resolver este problema es asignando una variable a cada una de las letras, obligando a que todas las variables tomen valores distintos y se cumpla la suma por lo que las restricciones son:

- $10^3(s+m) + 10^2(e+o) + 10(n+r) + d + e = 10^4m + 10^3o + 10^2n + 10e + y$
- restricción de todas las variables diferentes:$\neq (s, e, n, d, m, o, r, y)$
- El cero no puede ser el primer dígito de los tres números: $0 \neq (s, m)$

Modelando esto en Curry, se obtiene el siguiente programa. Se importa el módulo de *CLPFD*[3] para facilitar la codificación del problema:

```
import CLPFD

suma l =
        l =:= [s,e,n,d,m,o,r,y]
        & domain l 0 9
        & allDifferent l
        &       1000 *# s +# 100 *# e +# 10 *# n +# d
                +#
                1000 *# m +# 100 *# o +# 10 *# r +# e
                =#
                10000 *# m +# 1000 *# o +# 100 *# n +# 10 *# e +# y
        & s ># 0
        & m ># 0
        & labeling [] l
        where s,e,n,d,m,o,r,y free
```

Dando como única solución:

```
suma> suma [s,e,n,d,m,o,r,y] where s,e,n,d,m,o,r,y free
Free variables in goal: s, e, n, d, m, o, r, y
Result: success
Bindings:
s=9
e=5
n=6
d=7
m=1
o=0
r=8
y=2
More solutions? [Y(es)/n(o)/a(ll)] a
No more solutions.
```

3 Conclusión

Curry es un lenguaje muy completo, resultado de la mezcla de los paradigmas que lo componen. Esto permite que se resuelvan los problemas de forma más sencilla ya que el programador puede modelar su código de forma

muy similar a la realizadad. El implementar concurrencia en Curry es muy fácil gracias al uso de restricciones combinado con el operador "&" ya que el programador no tiene que agregar código extra y si se ejecuta en un equipo multinucleo adquiere la característica de paralelo. El inconveniente de esta facilidad es que el problema a resolver tiene que modelarse enfocado a restricciones para aprovechar la concurrencia. Pienso que es un lenguaje que está en crecimiento por lo que puede adherir nuevas características y funcionalidades para implementar concurrencia aprovechando las características de los paradigmas que lo conforman.

4 Agradecimientos

Agradezco a Fabián Maciel por su ayuda en la revisión de este artículo y a mi padre por sus consejos en el momento preciso.

Notas

[1] Símbolos o valores literales.

[2] *Portland Aachen Kiel System Curry*, que es una implementación de Curry basada en Prolog.

[3] Biblioteca de Curry para resolver restricciones de dominio finito.

Referencias

[1] Baber, F. & Salido, M. *Problemas de Satisfacción de Restricciones (CSP).* McGraw-Hill, 2008

[2] Hanus M. *Curry Report* http://www-ps.informatik.uni-kiel.de/currywiki/documentation/report Accedido el 30 de octubre del 2012.

[3] Hanus M. *Multi-paradigm Declarative Languages* http://www.informatik.uni-kiel.de/∼mh/papers/ICLP07.html Accedido el 30 de octubre del 2012.

[4] Hanus M. *Portland Aachen Kiel System Curry* http://www.informatik.uni-kiel.de/∼pakcs/ Accedido el 30 de octubre del 2012.

[5] Hanus M. *Tutorial on Curry* http://www-ps.informatik.uni-kiel.de/currywiki/documentation/tutorial Accedido el 30 de octubre del 2012.

[6] Vidal G. et al. *Técnicas de Fragmentación de Programas Multi-Paradigma.* http://users.dsic.upv.es/ gvidal/german/mist/tecfram.html Accedido el 30 de octubre del 2012.

Concurrencia en D

Fabián Maciel (A00967153) Román Villegas (A00967328)
Instituto Tecnológico y de Estudios Superiores de Monterrey
Campus Estado de México
Atizapán de Zaragoza, Estado de México, México.

31 de octubre, 2012.

Resumen

En los últimos años hemos visto un interesante surgimiento de bibliotecas y lenguajes de programación hechos para facilitar la realización de programas concurrentes. D es un lenguaje de programación que parte de la base de C++ agregando funcionalidad de otros paradigmas de programación; entre ellos la facilidad de crear programas concurrentes utilizando como herramienta principal el paso de mensajes.

1 Introducción

D es un lenguaje de sistemas que surge como una mejora práctica de C++, pero enriquecido de muchas maneras por otros lenguajes. Fue diseñado desde su incepción para ser multiparadigma, pues soporta la programación orientada a objetos, funcional, imperativa, concurrente y la metaprogramación. En este artículo se expondrá una breve introducción a D y se discutirá su enfoque en la concurrencia.

El lenguaje está interesado en los siguientes puntos:

- Desempeño. D fue pensado para ser un lenguaje de sistemas, por lo que se puede acceder a todas las capacidades de la máquina y programar sistemas operativos, controladores y aplicaciones. Tiene un modelo de memoria estructurado y compatible con C.
- Expresividad. El código en D es fácil de interpretar y entender en sus construcciones.
- Concurrencia. D se aleja de la manera en que lenguajes similares la manejan. En lugar de tener un sistema basado en memoria compartida implícita, utiliza *threads* independientes que se pueden comunicar por paso de mensajes.
- Código genérico. D integra poderosos mecanismos de mecanismos genéricos y generacionales para manipular código.
- Eclecticismo. D integra diferentes paradigmas de programación.

Dada la similitud que D tiene con sus lenguajes hermanos C y C++, se hará una descripción general del lenguaje haciendo comparaciones pertinentes. Programar en D resulta una transición natural y sencilla desde estos lenguajes.

2 D en acción

2.1 Similitudes con C/C++

D comparte una base reconocible de sentencias de C separadas por `;` y utilizando llaves como parte del paradigma imperativo con condicionales `if` y `switch`, ciclos `while`, `for` y `do while`. Maneja variables de tipo valor como estructuras (`struct`), enumeraciones (`enum`), uniones (`union`), apuntadores y los tipos primitivos numéricos, carácter, booleano y `void`. A esta lista, no obstante, agrega unos cuantos más como el tipo `function` y `delegate` para funciones normales y funciones que capturan variables, `string` (alias de `immutable(char)[]`), `real` y `dchar` (carácter tipo UTF32).

Las funciones se declaran de manera similar al recibir parámetros y regresar un tipo de valor. Los bloques también se manejan con llaves, haciendo que visualmente guarde mucha similitud con C. Cabe destacar que D también es un lenguaje con tipos estáticos.

En comparación con C++, se puede encontrar el concepto de alias para referirse a la misma variable con otro nombre. Además, comparten el paradigma orientado a objetos aunque con un acercamiento diferente por el uso de herencia simple e implementación de interfaces.

2.2 Diferencias y adiciones a C/C++

Una gran diferencia con sus lenguajes hermanos es la aparición del paradigma funcional. D soporta expresiones lambda, funciones de orden superior, inmutabilidad, *pattern matching*, *closures* y facilita la creación de funciones puras (funciones que garantizan que no existen efectos secundarios).

D permite definir la manera en que se comportan los parámetros de las funciones, ya sea para pasarse por referencia, de entrada o de salida con `ref`, `in` y `out`. Además de la manera común en que se pasan argumentos a las funciones con el uso de paréntesis, se puede incluir un conjunto más de paréntesis precedidos por un *!* justo después del nombre de la función para mandar argumentos de tiempo de compilación (a diferencia del segundo conjunto que se evalúan a tiempo de ejecución). Más adelante se menciona un uso importante de este tipo de parámetros.

Además de tener arreglos, añade diccionarios a los que denominan arreglos asociativos, en donde se relacionan valores con sus respectivas llaves. Éstos cuentan con verificación de límites (comenzando en índice 0), además de que conocen su longitud y pueden utilizar el carácter "$" para lograrlo. Si se necesita hacer uso de arreglos como son manejados en C, se puede utilizar el apuntador del arreglo accesible a través de `.ptr` para hacer aritmética de apuntadores sin que se tengan que respetar los límites. Igualmente se puede utilizar una opción de compilador para deshabilitar esta verificación. Los rangos pueden definirse fácilmente con `x .. y`, en donde el primer valor es inclusivo y el segundo exclusivo. Uno de sus usos más comunes es en *array-slicing*, que define un subconjunto del arreglo sin tener que definir ningún tipo de copia; ideal para algoritmos de divide y conquista recursivos.

El lenguaje añade semántica que es práctica en muchos casos y que hace que el código sea más fácil de entender. Por ejemplo, las palabras reservadas `is` e `in`. La primera apoya en la evaluación de tipos a tiempo de ejecución, mientras que la segunda apoya a los arreglos asociativos al preguntar si un dado valor existe. Introduce también una manera fácil de iterar con `foreach`, que puede moverse sobre los valores de un arreglo con o sin índice, los elementos de un arreglo asociativo con o sin su llave asociada.

Una característica que ayuda a la codificación y que simplifica algunas expresiones es que D tiene un sistema de inferencia de tipos, por lo que no es necesario especificarlos siempre. Esto no quita que el compilador haga verificaciones firmes de los tipos en los programas. Además, agrega el tipo `Variant` (definido en `std.variant`) que puede contener cualquier tipo de valor. `Variant` es un candidato ideal para utilizarlo como valor de regreso o de parámetros de métodos dinámicos.

Como parte de la metaprogramación, D incluye un concepto llamado `mixin` que sirve para evaluar y agregar código a tiempo de compilación, además de sentencias `static if` que sirven como condicionales para que

el compilador discrimine cuáles secciones de código deben de ser generadas. También incluye una manera intuitiva de generar plantillas, que son funciones que igualmente corren a tiempo de compilación y que hacen uso de lo descrito anteriormente para ser evaluadas con argumentos de compilación (utilizando *!* y paréntesis).

Un cambio muy importante en D es la facilidad y seguridad que ofrece en el manejo de la memoria. Ofrece un recolector de basura que se encarga de liberar memoria que ya no está siendo utilizada sin necesidad de preocuparse por hacerlo de manera manual. No obstante, la biblioteca estándar de D incluye la estándar de C, por lo que el programador tiene la flexibilidad de manejar la memoria al alocar y liberar manualmente. Una manera más en donde se puede especificar la liberación de memoria es con la sentencia `scope`. Definiendo esta sentencia con una salida normal o con una falla, se puede ejecutar código que maneje de manera adecuada la memoria utilizada. Por otro lado, en el manejo de errores D hace uso de excepciones y las maneja con sentencias `try`, `catch`, `finally` y `throw` como sucede en otros lenguajes como C# o Java.

El recolector de basura fue escrito en D, hecho que apoya a la definición de D como un lenguaje de sistemas. Si el programador desea hacer llamadas de más bajo nivel, D ofrece sentencias `asm` que permiten incluir código ensamblador de manera directa.

Siguiendo la línea de seguridad, D agrega el concepto de `final switch`. Cuando éste es utilizado con enumeraciones, el compilador revisa que todos los casos se hayan contemplado para que si algún programador añade un valor a la enumeración, se le avise que puede haber valores que no están siendo considerados en el `switch`.

D permite revisar validez de los datos en las operaciones a tiempo de ejecución utilizando contratos que pueden implementarse a través de *assertions*, precondiciones, postcondiciones e invariantes.

2.3 Inmutabilidad

Al incluir el paradigma de concurrencia, D ofrece la habilidad de definir variables inmutables. Utilizar el modificador `immutable` en una variable le dice al compilador que está prohibido cambiar el contenido de ésta en cualquier operación.

Este modificador permite el uso de paréntesis para definir exactamente qué es inmutable y qué no lo es. *immutable(char) [] str* define a los carácteres individuales como inmutables, pero no a `str`. *immutable char[] str* define todo como inmutable, es decir que `str` no puede cambiar a apuntar a otro arreglo.

La inmutabilidad ofrece garantías para compartir datos a través de *threads* de manera eficiente.

2.4 Transitividad

Un concepto importante dentro de la inmutabilidad es que ésta se transfiere de manera natural a todos los miembros de una variable cuando se utiliza este modificador. Pero, ¿qué sucede cuando hay indirección en un miembro de una variable? En el diseño de D se eligió utilizar transitividad en la inmutabilidad de todos los miembros, por lo que cualquier dato que pueda ser alcanzado desde una variable inmutable debe de ser inmutable también, es decir, toda la red de datos interconectados a ese valor a través de refs, arreglos y apuntadores.

D eligió este diseño gracias a su soporte de los principios de programación funcional y concurrente. La transitividad en la inmutabilidad le da la oportunidad al programador de utilizar el estilo funcional al mismo tiempo que el compilador puede verificar que este código no cambie datos inadvertidamente. Además, compartir datos inmutables entre *threads* es correcto, seguro y eficiente. Garantizar la transitividad impide que la inmutabilidad sea violada.

3 D avanzado

3.1 Concurrencia

Siendo D un lenguaje de sistemas, se ofrece una variedad de formas para crear programas concurrentes. A continuación se mencionan las formas y herramientas incluidas en el lenguaje.

La forma principal y sugerida por D es la utilización de *threads* aislados que se comunican a través de paso de mensajes. Sin embargo, también se provee sincronización de las conocidas secciones críticas protegidas por *mutexes* y variables de evento. Cualquier uso de operaciones o funciones que no se consideren seguras (a través de la propiedad `@safe`) es responsabilidad del programador.

3.2 No Compartir (por omisión)

Las variables en D, por omisión, no están compartidas. Se puede cambiar este comportamiento agregando el modificador `shared` antes de la variable para avisarle al compilador que se pretende compartir su valor y que se tomarán medidas especiales para realizar modificaciones.

```
int number; //no compartida
shared int sharedNumber; //compartida
```

Cada *thread* tiene su propia copia de las variables, pero se pueden comunicar entre ellos mediante el paso de mensajes asíncronos.

3.3 Creación de *threads*

Para inicializar un *thread* se utiliza la función `spawn` que recibe la dirección de la funcion `&fun` y el número de argumentos a1, a2, ..., a3. El número y tipo de argumentos debe coincidir con el de la función.

Ejemplo:

```
import std.concurrency, std.stdio;
void main() {
    auto low = 0, high = 100;
    spawn(&fun, low, high);
    foreach (i; low .. high) {
        writeln("Main thread: ", i);
    }
}

void fun(int low, int high) {
    foreach (i; low .. high) {
        writeln("Secondary thread: ", i);
    }
}
```

3.4 Compartición inmutable

Utilizando los conceptos anteriores de inmutabilidad y transitividad, resulta más sencillo comprender que cualquier variable inmutable puede ser compartida explícitamente entre diferentes *threads*. Cada que se crea

un nuevo *thread*, los argumentos que se le pasan deben de ser por valor y nunca por referencia (como podría ser el caso de arreglos) a excepción de cualquier variable inmutable. Está garantizado que cada que se acceda a su valor, éste no va a ser diferente bajo ninguna circunstancia. No hay necesidad de poner más controles para asegurar que el programa correrá de manera segura gracias a la labor del compilador por asegurarse de que no puede haber modificaciones en una variable inmutable ni en sus miembros.

3.5 Intercambio de mensajes entre *threads*

Para que un *thread* se pueda comunicar con otro mediante el paso de mensajes necesita de una forma de referirse al *thread* al que le quiere mandar el mensaje. El envío de mensajes en D se realiza mediante el envío de información utilizando la dirección del *thread* al que se le quiere mandar la información.

La dirección de un *thread* es de tipo `Tid`. `spawn` regresa el `Tid` del *thread* creado y la propiedad global `thisTid` regresa el `Tid` del *thread* que se está ejecutando.

Para mandar un mensaje se utiliza la función `send`, que recibe la dirección del *thread* a enviar y los parámetros que se quieren enviar. Para recibir un mensaje se utiliza la función `receive`.

3.6 Formas de recibir

3.6.1 receiveOnly!tipoEspecífico();

Esta función sólo acepta tipos específicos, por ejemplo:

```
receiveOnly!bool(); //sólo acepta booleanos
receiveOnly!(Tid, int)(); //sólo recibe un Tid junto con un entero
```

3.6.2 *Pattern matching* con *receive*

La función de `receive` puede escribirse de manera que lo que recibe coincida con lo que se desea hacer para tener una funcionalidad personalizada.

La función `receive` recibe a manera de parejas lo que se desea manejar en forma de {(tipo nombreVariable){ cuerpo del método }}

```
receive(
    (string s) { writeln("Got a string with value ", s); },
    (int x) { writeln("Got an int with value ", x); }
);
```

Nótese que cada cláusula está separada por una coma y al final no se incluye ninguna. Otra cosa a considerar es que al enviar un mensaje, este coincidirá con el primer patrón que se encuentre dentro de la función.

```
receive(
    (long x){ ... }
    (string x){ ... }
    (int x){ ... }
);
```

Este código no compila, pues la sección de `(int x)` nunca será evaluada porque todos los números serán atrapados en la sección de `(long x)`.

Para hacer coincidir con cualquier mensaje se puede utilizar el tipo de variable `Variant` de esta manera: `(Variant any) { ... }`.

3.7 Terminación de *threads*

Para manejar la terminación de *threads*, D provee un mecanismo de *owner/owned* en el que el *thread* que crea a otro es el dueño y el *thread* creado es el adueñado. Se puede cambiar dinámicamente el dueño usando la función `setOwner(tid)`. La relación no es necesariamente unitaria y entre dos *threads* puede existir la relación *owner/owned* en donde el primero que termine le notificará al segundo. Un factor a considerar es que cuando el dueño termine su ejecución, las llamadas de receive al *thread* adueñado lanzarán una excepción. Sin embargo, todas las llamadas hechas previamente a `receive` sin terminar se completarán aunque el dueño ya haya terminado.

Cuando el dueño termina con una excepción, es importante que se informe a los *threads* adueñados que hubo un error. Esto se realiza mediante mensajes fuera de banda utilizando la función `prioritySend` en lugar de `send`.

3.8 *Mailbox crowding*

Los *threads* reciben los mensajes en un buzón. Los buzones de cada *thread* tienen un tamaño límite que puede ser cambiado por el programador. Si en algún momento se excede su tamaño, D ofrece una manera de manejar la situación en un `enum` llamado `OnCrowding`, en dónde se escoge si se bloquean los mensajes entrantes, si se lanza la excepción o si se ignorarán los mensajes que entren.

3.9 *Sharing*

Para crear una variable compartida utilizamos

```
shared tipo variable;
```

Utilizar una variable compartida obliga al programador a ser más cuidadoso con las funciones usadas. El compilador también está consciente de esto y protege al programador al no permitirle hacer usos inadecuados sobre éstas, por ejemplo, al rechazar cualquier operación no atómica sobre los cambios. Para alterar atómicamente números, la biblioteca de `concurrency` de D provee el método `atomicOp` que recibe un string con la operación y la referencia al número a cambiar y el otro número de la operación. Es importante notar que todos los tipos en D pueden sufrir alteraciones de manera atómica, excepto en el tipo real que depende directamente de la implementación de la plataforma.

Es importante considerar que la propiedad `shared` es transitiva y que variables con este modificador pueden ser compartidas vía las funciones `send` y `receive`.

Otro factor a considerar es que D garantiza la consistencia secuencial del código de manera que en el orden en el que se lee y escribe es el mismo que en el código dentro de un mismo *thread*. A nivel global, las lecturas y escrituras se perciben como entrelazadas por múltiples *threads*. Para poder garantizar que los cambios a las variables compartidas sean visibles por todos los *threads*, los accesos a éstas son realizados con instrucciones de máquina especiales llamadas barreras de memoria. Realizar esta serialización es lento y caro y el compilador no puede hacer muchas optimizaciones que en ocasiones incluyen reordenamiento de instrucciones. El diseño de D justifica esto porque el uso de variables compartidas es reducido y sugiriendo utilizar mejor copias locales en cada *thread* y solamente escribir en la compartida una vez finalizado su proceso.

D ofrece nivel de sincronización tradicional pero limitada intencionalmente desde su diseño, ya que lo hace a nivel de clase con el modificador `synchronized`. Este tipo de sincronización está basado en el uso de candados que serializan el acceso a todos los métodos de una clase. Las clases con el calificativo `synchronized` tienen ciertas características:

- No puede haber datos públicos.
- Todos los métodos son sincronizados.
- El acceso a elementos protegidos es restringido a miembros de la clase y sus decendientes.
- El acceso a elementos privados está restringido a miembros de la clase.
- El compilador protege a los miembros para que no escapen al restringir pasos por referencia.

Un último punto a considerar es que se puede quitar el *cast* de `shared` en cualquier momento y que al usar métodos sincronizados se deben tomar en cuenta *deadlocks* y otros problemas relacionados con la sincronización tradicional.

3.10 Ejemplo: Cálculo de Pi

En este sección se pueden observar dos programas en D que hacen el cáluclo de Pi utilizando la fórmula:

$$\pi = \int_0^1 \frac{4}{1+x^2} dx$$

El siguiente código muestra el cálculo de Pi de manera secuencial. Como se puede apreciar, visualmente es muy parecido a sus lenguaje hermano C.

```
import std.stdio;

void main() {
    auto num_rects = 100000L;
    double mid, height, width, area;
    double sum = 0.0;

    width = 1.0 / cast(double) num_rects;
    for (long i = 0; i < num_rects; i++) {
        mid = (i + 0.5) * width;
        height = 4.0 / (1.0 + mid * mid);
        sum += height;
    }
    area = width * sum;
    writefln("Computed pi = %.20f\n", area);
}
```

Esta segunda versión del cálculo se realiza de manera concurrente. Se decidió utilizar creación de *threads* y paso de mensajes en lugar del tradicional método de sincronización.

```
import std.stdio, std.concurrency;

// Recibe el identificador del padre, los límites de ejecución de esta parte
// y el ancho del rectángulo del algoritmo
void piPiece(Tid dad,long start, long end, double width){
    double sum = 0.0;
    foreach(i; start .. end){
            double mid = (i + 0.5) * width;
            double height = 4.0 / (1.0 + mid * mid);
            sum += height;
    }
    // envía el valor de la suma al padre utilizando paso de mensajes
    send(dad, sum);
}

void main() {
    auto num_rects = 100000L;
    double mid, height, width, area;
    double sum = 0.0;
    width = 1.0 / cast(double) num_rects;

    // el número de partes en que se quiere dividir el problema
    long veces = 100;
    long pedazo = num_rects / veces;
    for (long i = 0; i < veces; i++) {
            // se crea el thread con los parámetros que espera la función piPiece
            auto tid = spawn(&piPiece, thisTid, i*pedazo, pedazo + i*pedazo, width);
    }

    // se atienden los mensajes que cada parte del cálculo
    // regresa como paso de mensajes
    foreach(i; 0 .. veces){
            receive(
                    // patrón recibido en el mensaje
                    (double sumParcial) { sum += sumParcial; }
            );
    }
    area = width * sum;
    writefln("Computed pi = %.20f", area);
}
```

4 Conclusión

D es un lenguaje que nos ofrece una versión aumentada y mejorada de C++. Tenemos a nuestra disposición todas las herramientas para desarrollar cualquier aplicación que deseemos con un altísimo grado de control, teniendo al mismo tiempo la posibilidad de utilizar elementos del mismo lenguaje que nos facilitan ciertas tareas. Es agradable ver que D nos provee de soluciones que necesitan atención especial en C++ y que nos dan la ventaja de despreocuparnos de particularidades que sólo nos quitarían tiempo de desarrollo o pruebas.

De igual manera es interesante ver que la concurrencia manejada en D tiene un enfoque sumamente moderno, desafiando paradigmas de los lenguajes sobre los que está basado, ya que toma partes probadas de otros lenguajes para resolver problemas con diferentes paradigmas de programación. Esta flexibilidad e innovación

le da al programador diversas herramientas empaquetadas en un mismo lenguaje de programación para que no necesite disponer de bibliotecas a la hora de desarrollar aplicaciones o al enfrentarse a problemas.

Cabe destacar que D, al reunir lo mejor de diferentes lenguajes, requiere de un dominio de conceptos que van de nivel intermedio a avanzado de programación. La transición desde un lenguaje como C o C++ resulta natural y fluida, pero tener conocimientos de programación funcional y concurrente y de metaprogramación son los que desatan el verdadero potencial de un programador que utiliza D.

El manejo y soporte por parte de D para la concurrencia es bastante sofisticado, sobre todo por el rol tan importante que juega el compilador en la validación de código mientras se asegura de eliminar la mayor cantidad posible de problemas que pueden surgir al correr programas que utilizan el paralelismo. De las formas en las cuales D soporta la concurrencia, se recomienda más la utilización de variables explícitamente no compartidas en los *threads* y la comunicación entre ellos por paso de mensajes. En caso de compartir datos, es recomendable que se haga uso de variables inmutables.

Referencias

[1] *Statements.*
http://dlang.org/statement.html Accedido el 27 de octubre del 2012.

[2] Alexandrescu, A. *The D Programming Language.* Addison-Wesley, 2010.

Lenguaje de programación Fortress y paralelismo

Andrés Hernando Márquez (A01164612) Carlos Mohedano Flores (A01165426)
Instituto Tecnológico y de Estudios Superiores de Monterrey
Campus Estado de México
Atizapán de Zaragoza, Estado de México, México.

31 de octubre, 2012.

Resumen

Este documento explica el lenguaje de programación *Fortress*, sus características principales, la forma en que maneja el paralelismo y los métodos para lograrlo.

1 Introducción

En el mundo de los lenguajes de programación existen cientos de ellos con diferentes características que los hacen únicos y que están principalmente desarrollados para cumplir con ciertas funciones para facilitar el trabajo a los profesionistas o personas que los usen. Uno de ellos es el lenguaje llamado *Fortress* el cual está diseñado para el cómputo de alto desempeño y tiene sus raíces en los lenguajes como Fortran, Scala y Haskell. *Fortress* fue creado por la ex-empresa Sun Microsystems con un apoyo económico del proyecto *DARPA's High Productivity Computing Systems*. En Marzo del 2008 el proyecto se vuelve un intérprete de referencia paralela de código abierto el cual es una implementación completa de la especificación 1.0. El objetivo principal que tenían los creadores del lenguaje era buscar ideas sobre el diseño de un gran lenguaje de programación y usarlas como propias para su nuevo lenguaje.

1.1 Generalidades

El lenguaje *Fortress* debe su nombre a la idea de ser un Fortran seguro que provee fuerte abstracción y seguridad en tipos según los principios de los lenguajes de programación modernos. Entre sus características principales se encuentran un paralelismo implícito para los ciclos más comunes y facilitando el trabajado de administrar los hilos de ejecución, soporte para caracteres *Unicode* y una sintaxis muy concreta similar a la notación matemática. *Fortress* está desarrollado para crear programas paralelos con facilidad combinando una gran funcionalidad con bibliotecas desarrolladas en Java pero optimizando todos los procesos.

1.2 Características

El lenguaje *Fortress* facilita a los programadores permitiendo insertar en código los requerimientos que cada método necesita para funcionar así como declarar la salida esperada por el programador.

```
factorial(n)
  requires {n>=0}
  ensures {result >= 0 } =
    result ZZ32 := 0
    if n=0 then result = 1
    else result = n factorial(n-1) end
```

Fortress permite guardar tipos definidos por el programador como las unidades de medida para prevenir errores como sumar kilómetros a una variable que está dada en millas o con cualquier otra unidad de medida.

```
distance := 60 miles/hour (3600 seconds in hours)
```

Una característica que tiene el lenguaje es la posibilidad de definir métodos para la sobre escritura de operadores que vayan a ser aplicados sobre objetos de la clase que elijamos sobrescribir el operador.

```
trait BigNum extends Number
 opr-(Number, self):BigNum
 ...

end
```

Fortress también soporta las definiciones de funciones sin y con recursión así como funciones mutuamente recursivas.

```
factorial(n) =
 if n = 0 then 1
 else n factorial(n-1)
end
```

El lenguaje *Fortress* se diseñó con la simple idea de ir creando al principio un núcleo pequeño y que con el tiempo se vayan escribiendo bibliotecas para que evolucione y vaya creciendo el soporte técnico teniendo a varias personas trabajando sobre él y que pueda ser como otros lenguajes modernos y grandes y pueda ser utilizado para resolver grandes problemas.

1.3 Tipos de datos

El lenguaje cuenta con los tipos de datos comunes para los demás lenguajes como las cadenas de texto, los valores de verdadero o falso (*Booleans*) y los numéricos. La diferencia de *Fortress* con los demás lenguajes es la sintaxis para representarlos, ya que la sintaxis de un número de punto flotante es RR64 para los de precisión de 64 bits mientras que los de 32 bits es RR32, al igual que los números enteros se escriben ZZ32 o ZZ64.

Estos tipos de datos representan los conjuntos de los números matemáticamente, los enteros (ZZ) y los reales (RR) y se compila con el siguiente formato: $\mathbb{Z}$ y $\mathbb{R}$ respectivamente.

2 Sintaxis

El lenguaje *Fortress* se caracteriza principalmente por el estilo matemático en su sintaxis, ya que lo que busca es emular la notación matemática mejorando al lenguaje Fortran en ese sentido. Por ejemplo los nombres de variables se compilan a un estilo cursivo y así existen diferentes reglas de diseño para cada parte de un programa *Fortress*. El operador

```
^
```

se utiliza en este caso para denotar potencia y se compila a superíndices y los símbolos

```
[ y ]
```

se compilan a subíndices:

```
f(x) = x^2 + sin x - cos 2 x
```

se compila a

$$f(x) = x^2 + \sin x - \cos 2x$$

y

```
    a[i]
```

se compila a

$$a_i$$

Para todas las funciones o elementos que son básicos en matemáticas hay una palabra reservada que a tiempo de compilación se genera el símbolo correspondiente para tener un mejor ambiente matemático y que los programadores (matemáticos o no) se sientan mas cómodos y sientan que están trabajando en papel como lo hacían antes pero ahora con la ayuda de las máquinas. También se utilizan combinación de caracteres para la emulación de los distintos elementos de la notación matemática. La meta que se busca con el lenguaje *Fortress* es que los programadores escriban el código como si estuvieran trabajando en un pizarrón o en una hoja.

A continuación se muestra la tabla con las equivalencias de las palabras reservadas con su respectivo símbolo matemático:

palabra	*simbolo*	*palabra*	*simbolo*
BY	$\times$	TIMES	$\times$
DOT	$\cdot$	CROSS	$\times$
CUP	$\cup$	CAP	$\cap$
BOTTOM	$\perp$	TOP	$\top$
SUM	$\sum$	PROD	$\prod$
INTEGRAL	$\int$	EMPTYSET	$\emptyset$
SUBSET	$\subset$	NOTSUBSET	$\not\subset$
SUBSETEQ	$\subseteq$	NOTSUBSETEQ	$\not\subseteq$
EQUIV	$\equiv$	NOTEQUIV	$\not\equiv$
IN	$\in$	NOTIN	$\notin$
LT	$<$	LE	$\leq$
GT	$>$	GE	$\geq$
EQ	$=$	NE	$\neq$
AND	$\bigwedge$	OR	$\bigvee$
NOT	$\neg$	XOR	$\bigoplus$
INF	∞	SQRT	$\surd$

3 Constructores primitivos de paralelismo

Otro punto muy importante en *Fortress* es que está desarrollado con paralelismo como un estándar para las operaciones que se vayan a realizar y así aprovechar mejor los recursos que las máquinas poseen como varios procesadores. El método que se usa en el lenguaje para implementar el paralelismo es implícito y trabaja por robo de tareas, es decir a cada procesador o núcleo de procesador se le asigna una tarea en específico que tiene que realizar sobre cierta información y cuando termine puede revisar la carga de trabajo de los otros procesadores y tomar tareas que están en una fila de espera para resolverlas y así terminar en un menor tiempo. Los desarrolladores al crear el lenguaje no vieron a la programación en paralelo como una meta que tenían que llegar, sino como un compromiso pragmático que debían resolver para tener un mejor lenguaje de programación. En el lenguaje *Fortress* los ciclos son paralelos por default y se crean tantos hilos de ejecución como sean necesarios de forma automática.

3.1 Comando for

El ciclo más usado en *Fortress* es el for y tiene la siguiente forma:

```
for i <- 1:10 do
 print i
```

Lo que hace la línea de código anterior es crear 10 hilos y a cada uno le asigna el valor de la i correspondiente con la instrucción de imprimir. Un punto a recalcar aquí es que la salida no es determinística ya que no se sabe el orden en que correrán los hilos y los números del 1 al 10 pueden salir en diferente orden en cada corrida.

3.2 Tuplas

Las tuplas son una estructura de datos donde se crean hilos en automático cada vez que se crea uno y el número de hilos creados es el número de elementos que contenga la tupla.

```
(a1,a2,a3) = (e1,e2,e2)
```

En el caso anterior, se crearían tres hilos, cada uno asignando a las a's los valores de las e's.

3.3 Hilos explícitos

Así como se contruyen hilos de forma automática gracias a los ciclos, los generadores y las tuplas, el programador tiene la opción de crear sus propios hilos para la tarea que más le convenga y la forma de hacerlo es muy simple:

```
t1 = spawn do e1 end
t2 = spawn do e2 end
a1 = t1.value()
a2 = t2.value()
```

En el código anterior se crean dos hilos y en ese momento empiezan a correr. Aun cuando existe la posibilidad de crear hilos de esta forma, no es recomendado.

3.4 Sentencia do...also do

Hay una última forma para indicarle a la máquina virtual las actividades que queremos que corran en paralelo y es con el comando do also do. Este comando es útil cuando sabemos la cantidad exacta de las operaciones que se realizarán en paralelo y que son independientes entre sí para que no haya conflictos.

```
component prog2
export Executable

factorial(n)
  requires {n>=0}
  = if n=0 then 1
    else n factorial(n-1) end

run () =
  do
    factorial(100)
  also do
    factorial(500)
  also do
    factorial(1000)
  end
end
```

El ejemplo anterior señala que se correrá la función `factorial` tres veces con tres argumentos distintos pero cada uno en un hilo de ejecución separado. Esta forma permite cualquier número de `also do` pero por lo menos debe existir la instrucción `do` y el `end` al final.

4 Generadores y reductores

Una cualidad de paralelismo del lenguaje *Fortress* son los generadores. Los generadores son objetos encargados del manejo de iteraciones, paralelismo y asignación de los hilos de ejecución a los procesadores. Dado

que el estándar, de cierta forma, en *Fortress* es el paralelismo, los generadores no son la excepción, aunque tienen métodos secuenciales; obviamente usar estos métodos no es la mejor práctica pues sería como tener un auto deportivo y sólo manejar tu viejo vocho. Los Reductores son expresiones que se encargan de *juntar* diferentes resultados de otras operaciones o los valores devueltos de algún generador. Algunas funciones ejemplo serían, suma o máximo. Para dejar más claro el concepto de generadores y reductores se podría decir que ambos trabajan de forma similar que las funciones map y reduce, donde map sería un Generador y reduce un reductor, aunque obviamente con un comportamiento paralelo por defecto.

```
object SumZZ32 extends Reduction [[ZZ32]]
    empty():ZZ32 = 0
    join(a:Z32, b:Z32):Z32 = a + b
end
z = (1 # 100).generate[[ZZ32]] (SumZZ32, fn (x) ) 3x + 2)
```

En el ejemplo anterior, se ha declaro un reductor llamado SumZZ32 que simplemente representa la operación 3x + 2 donde x va de 1 a 100. Recordar que lo anterior se ejecuta de forma paralela.

5 Bloques atómicos

Como en muchos otros lenguajes de programación que soportan procesos paralelos o hilos de ejecución es inevitable que surjan problemas de paralelismo (p.ej. condición de carrera). En el caso de *Fortress* podríamos decir que este tipo de problemas es en cierta medida más común que sucedan debido a como se explicó anteriormente, *Fortress* implementa ciclos en forma paralela de forma implícita. El siguiente ejemplo de código en *Fortress* realiza la suma de los cuadrados de los números en una lista dada. Al parecer no habría problema para compilar y hacer pruebas del programa y es verdad, el programa compila y ejecuta sin problemas, sin embargo, como se mencionó anteriormente debido a que los ciclos en *Fortress* se manejan de forma paralela de manera implícita, en este ejemplo en particular se da el problema de condición de carrera para la variable sum.

```
sumOfSquares( n:List[\ZZ32\] ) : ZZ64  = do
    sum ZZ64 := 0
    for i<-0#|n| do
        sum += (n[i])^2
    end
    sum
end

run ():() = do
    theList = <|[\ZZ32\] x| x<-1#100|>
    println sumOfSquares =
    sumOfSquares(theList)
end
end
```

La solución es muy sencilla, simplemente recurrimos a la palabra reservada *atomic* para crear un bloque atómico que como ya sabemos, en un bloque atómico se ejecuta todo exitosamente o no se ejecuta nada.

```
sumOfSquares( n:List[\ZZ32\] ) : ZZ64  = do
    sum ZZ64 := 0
    for i<-0#|n| do
      atomic  sum += (n[i])^2
```

```
    end
    sum
end

run ():() = do
    theList = <|[\ZZ32\] x| x<-1#100|>
    println sumOfSquares =
    sumOfSquares(theList)
end
end
```

6 Futuro de Fortress

Al parecer *Fortress* no tendrá un futuro muy prometedor pues el pasado 20 de julio de 2012 Oracle anunció en su blog el cierre del proyecto. El anuncio fue hecho por Guy Steele, miembro del Laboratorio de Investigación de Lenguajes de Programación de Oracle.

El proyecto *Fortress* llevaba ya casi diez años de diseño, desarrollo e implementación, y según Steele ese periodo de tiempo es bastante largo para una investigación de la industria, lo normal sería un periodo entre uno y tres años, pero aun así, Steele considera que fue un periodo de tiempo que valió la pena. De acuerdo a Steele, el motivo principal del cierre del proyecto es por la cantidad de problemas técnicos que se encontraron al intentar implementar un compilador enfocado a la JVM, la cual no está diseñada para soportar el sistema de tipos de *Fortress*. Pero algo interesante que declara Steele en su publicación es que prácticamente, lo que se tenía que aprender lo habían hecho ya, y terminar la implementación de un compilador para *Fortress* en la JVM no conllevaría a más aprendizaje, en el sentido de investigación. Además de esta justificación, Steele señala que otros lenguajes (como Clojure o Scala) han experimentado los mismos problemas que *Fortress* durante los últimos 10 años. Pero aunque *Fortress* se ha quedado sin soporte de Oracle, el proyecto ha quedado como abierto, y durante estos meses la documentación se piensa dejar lo más accesible y completa posible, además de que se arreglarán *bugs* pero sólo si es requerido por los usuarios. Sobre lo anterior debemos recordar que a final de cuentas Oracle es una organización que genera ganancias y básicamente el proyecto *Fortress* no le generaba ninguna, tal vez esta sea la mayor, y por mucho, la razón por la que cerraron el proyecto.

7 Conclusión

El desarrollo de este trabajo nos ha ayudado a expandir nuestro conocimiento sobre lenguajes de programación y los diferentes alcances en lo que refiere a la programación concurrente y paralela sobre la forma que los autores la ven y diseñan. Cuando vimos por primera vez un programa escrito en *Fortress* nos pareció algo extraño ver la notación matemática en un programa computacional debido a que no estamos acostumbrados a ello. Estudiándolo nos dimos cuenta que al crear de esa manera el lenguaje, la forma de programar es más intuitivo para los matemáticos. Ya no diseñarían y analizarían sus problemas en papel, ahora sería en una computadora. Por el otro lado, sentimos que al lenguaje le faltó más formas de expansión para darse a conocer en todos los rubros porque la idea de que el paralelismo sea una característica por defecto lo hace interesante y el manejo de hilos de ejecución es muy sencillo. A pesar de que Oracle no continuará con más investigación y desarrollo del lenguaje pensamos que varias características de este lenguaje deberían ser tomadas en cuenta para el desarrollo de futuros lenguajes de programación.

8 Agradecimiento

Queremos agradecer a nuestro profesor Ariel Ortiz Ramírez en la enseñanza sobre programación multinúcleo y por este trabajo que tuvimos que investigar sobre un lenguaje nuevo.

Referencias

[1] Allen, Eric. et.al. *The Fortress Language Specification.*
Sun MicroSystems, 31 de Marzo del 2008.

[2] H. Flood, Christine. *Project Fortress: a new programming language from sun labs*
Sun Microsystems Laboratories, JavaOne Conference 2008.

[3] Steele, Guy. Maessen, Jan-Willem. *Fortress Programming Language Tutorial.* Sun Microsystems Laboratories, 11 de Junio de 2006.

Programación multinúcleo utilizando F#

Manuel González Solano (A01165461) Felipe Donato Arrazola Gómez (A01165547)
Instituto Tecnológico y de Estudios Superiores de Monterrey
Campus Estado de México
Atizapán de Zaragoza, Estado de México, México.

31 de octubre, 2012.

Resumen

El presente documento tiene como objetivo analizar F#, lenguaje de programación multiparadigma de la plataforma .NET, para la realización de aplicaciones concurrentes. Se analizará también el uso de `Thread` y de `BackgroundWorker` para lograr paralelismo y concurrencia en F#, y se cubrirán los patrones de diseño utilizados en F#, *Asynchronous Programming Model* (APM) y *Asynchronous Workflows*, usados para explotar las capacidades concurrentes del lenguaje.

1 Introducción

F# es un lenguaje de programación que opera sobre la plataforma .NET e incluye paradigmas como programación funcional, así como también programación imperativa y programación orientada a objetos. F# es una variante de ML y es compatible con implementaciones de OCaml. F# fue originalmente desarrollado por Don Syme en los Laboratorios de Investigación de Microsoft en Cambridge y actualmente se distribuye como un lenguaje totalmente soportado en la plataforma .NET y en Visual Studio.

F# es un lenguaje que utiliza inferencia de tipos y además soporta declaraciones de tipos explícitas. F# soporta todos los tipos que están dentro del Common Language Infrastructure (CLI) y además categoriza esos tipos como inmutables, lo cual facilita el diseño de aplicaciones multinúcleo, o como mutables. F# es un lenguaje de programación simple y pragmático que tiene fortalezas particulares en programación orientada a datos, programación paralela de operaciones de entrada/salida, programación paralela en CPU, scripting y desarrollo de algoritmos. Además permite el acceso a una gran biblioteca de herramientas base ya incluidas en Visual Studio.

1.1 Conceptos básicos relevantes

Para que un lenguaje de programación sea considerado como fucional, típicamente el lenguaje debe de soportar algunas funcionalidades específicas:

- Datos inmutables.
- Habilidad para componer funciones.
- Que las funciones puedan ser tratadas como datos.
- Evaluación diferida (mejor conocida como *lazy evaluation*).
- Coincidencia de patrones (mejor conocida como *pattern matching*).

F# provee de diversas construcciones y ciertos tipos de datos inmutables como: tuplas, listas, uniones discriminantes y registros.

Una tupla es una coleccion ordenada de datos y una manera fácil de agrupar pequeños trozos de información. Pueden ser usadas para rastrear resultados intermedios de cierto calculo.

```
> // tuplas
let comida = ("hamburguesa", "papas a la francesa", "pizza");;
val comida : string * string * string
```

Mientras que las tuplas agrupan valores en una sola entidad, las listas permiten ligar datos en forma de cadena.

```
> //listas
let numeros = [1; 2; 3; 4];;
val numeros : int list = [1; 2; 3; 4]
```

Las uniones discriminantes es un tipo de dato que sólo puede ser uno de un conjunto de valores posibles.

```
> // uniones discriminantes
type Pizza =
    | Hawaiiana
    | Peperonni
    | Pollo
    | Suprema;;

type Pizza =
    | Hawaiiana
    | Peperonni
    | Pollo
    | Suprema
```

Las uniones discriminantes son buenas para definir jerarquías, pero cuando se trata de obtener valores de ellas, tienen el mismo problema de las tuplas, no hay alguna asociación con cada valor. Los registros dan una forma de organizar valores en tipos y nombrarlos a través de campos.

```
> //registros
type Persona = { Nombre : string; Apellido : string; Edad : int };;
type Persona =
    {Nombre : string;
     Apellido : string;
     Edad : int;}
```

F# soporta la definición de objetos de la siguiente forma:

```
type Punto =
    val m_x : float
    val m_y : float
    // Constructor
    new (x, y) = { m_x = x; m_y = y }
    new () = { m_x = 0.0; m_y = 0.0 }
    member this.Length =
        let sqr x = x * x
        sqrt <| sqr this.m_x + sqr this.m_y
```

En este caso, `m_x` y `m_y` son atributos de la clase `Punto`, y `Length` es un método que cualquier objeto `Punto` puede invocar.

1.2 Expresiones Computacionales

Computation Expressions, o Expresiones Computacionales, en F# proveen de una sintaxis conveniente para escribir operaciones que pueden ser secuenciadas y combinadas utilizando construcciones de flujos de control y ataduras (*bindings*). Pueden ser utilizadas para dar una sintaxis conveniente para algunas *monadas* (*monads* en inglés), una característica de la programación funcional que se utiliza para manejar datos, control, y efectos secundarios (como entrada/salida) en programas funcionales. Otra forma de pensar en expresiones computacionales es que ellas permiten insertar código entre varios pasos de una operación, haciendo cualquier procesamiento sin requerir que explícitamente se escriba el código.

Las expresiones en secuencia son un ejemplo de expresiones computacionales, como lo son el flujo de trabajo asíncrono y las *query expressions*.

La sintaxis básica de las expresiones computacionales sigue la forma *builder-name* `{` *expression* `}`. Todas las expresiones computacionales se descomponen en múltiples funciones al constructor de expresiones. En expresiones computacionales, dos formas están disponibles para algunas construcciones comunes. Se puede invocar construcciones variantes utilizando el sufijo ! (bang) en ciertas palabras reservadas, como `let`!, `do`! y algunas más.

1.2.1 Definiendo expresiones computacionales

Se pueden definir características de las expresiones computacionales propietarias creando una clase constructora y definiendo ciertos métodos de esa clase. A continuación se muestran los métodos de una expresión computacional.

- `member For: seq<'a> * ('a -> Result<unit>) -> Result<unit>`: Permite la ejecución de ciclos for. Los parámetros son valores que el ciclo ejecuta en el cuerpo del ciclo for.
- `member Zero: unit -> Result<unit>`: Permite la ejecución de unidades de expresión, como el resultado de una expresión `if` sin un `else` que evalue a falso.
- `member Combine: Result<unit> * Result<'a> -> Result<'a>`: Utilizado para ligar partes de expresiones computacionales, como dos ciclos for en secuencia.
- `member While: (unit -> bool) * Result<unit> -> Result<unit>`: Permite la ejecución de ciclos `while`. Los parámetros de la función determinan cuando debería continuar el ciclo.
- `member Return: 'a -> Result<'a>`: Permite la ejecución de la palabra `return`.
- `member ReturnFrom: 'a -> Result<'a>`: Permite la ejecución de la palabra `return`!.
- `member Yield: 'a -> Result<'a>`: Permite la ejecución de la palabra `yield`.
- `member YieldFrom: seq<'a> -> Result<'a>`: Permite la ejecución de la palabra `yield`!.
- `member Delay: (unit -> Result<'a>) -> Result<'a>`: Esta operación se utiliza en conjunción con `Combine` para asegurar que las operaciones se ejecuten en el orden correcto (en caso de efectos secundarios).
- `member Run: Result<'a> -> Result<'a>`: De ser provisto, este método será llamado al principio de cada expresión computacional.
- `member Using: 'a * ('a -> Result<'b>) -> Result<'b> when 'a :> System.IDisposable`: Permite la ejecución de `use` y `use`!.

- `member Bind: Result<'a> * ('a -> Result<'b>) -> Result<'b>`: Permite la ejecución de `let!` y `do!`.
- `member TryFinally: Result<'a> * (unit -> unit) -> Result<'a>`: Permite la ejecución de `try/finally`. Los parámetros son el resultado del bloque `try` y de la función que representa el bloque `finally`.
- `member TryWith: Result<'a> * (exn -> Result<'a>) -> Result<'a>`: Permite la ejecución de `try/with`. Los parámetros son el resutado del bloque `try` y la función representada por el bloque `with`.

2 Paralelismo y concurrencia en F#

2.1 Cómo se logra el paralelismo y la concurrencia en F#

F# ofrece opciones de paralelismo, concurrencia y tareas asíncronas en el lenguaje, y esto lo logra a través del manejo de hilos. El concepto de hilos, como bien se describió en la sección anterior, es similar al que se viene manejando en otros lenguajes como C y en el ambiente de sistemas operativos, la diferencia siendo los métodos que esta clase tiene y cómo se utilizan.

2.2 Threads

El uso de hilos se logra usando la clase `System.Threading.Thread`. `Thread` toma como parámetro una función, ya sea definida o una función lambda la cual ejecutará el hilo en cuanto arranque. Existen tres funciones principales que `Thread` manda a llamar.

- `Start`
- `Sleep`
- `Abort`

`Start` se encarga de ejecutar al objeto `Thread`, y al hacerlo empieza a ejecutarse la función que recibió como parámetro. `Sleep` es un método estático que manda a dormir al objeto `Thread` por un periodo de tiempo. Finalmente, `Abort` intenta matar al objeto `Thread` lanzando una excepción de tipo `ThreadAbortException` [3].

El siguiente ejemplo muestra cómo se crea un hilo y se manda a ejecutar.

```
let threadBody() =
    for i in 1 .. 5 do
        Thread.Sleep(200)
        printfn "[Hilo con id: %d] %d..."
            Thread.CurrentThread.ManagedThreadId
            i

let spawnThread() =
    let thread = new Thread(threadBody)
    thread.Start()

spawnThread()
```

La función `spawnThread` crea un nuevo objeto `Thread`, pasándole como parámetro la función `threadBody`, y manda a ejecutarlo con la llamada a `Start`. La función `threadBody` itera del uno al cinco, imprimiendo el id del hilo y el número de iteraciones que lleva.

El uso de hilos directamente para implementar paralelismo y concurrencia en un programa tiene más desventajas que puntos a favor; aunque le otorgan al usuario un alto grado de control, cuando se trata de paralelizar un programa esto no siempre es la mejor solución. Cada hilo tiene su propia pila que puede alcanzar un tamaño de varios megabytes lo cual implica que la creación innecesaria de estos objetos puede ser muy costosa. Por lo tanto, las bibliotecas de .NET ofrecen una fuente de hilos que está disponible sin necesidad de crear un hilo nuevo.

2.2.1 `ThreadPool`

`ThreadPool` es un conjunto de hilos ya creados y disponibles para ser utilizados por el usuario. Para mandar a pedir un nuevo hilo, se invoca el método `QueueUserWorkItem` el cual toma como parámetro una función que será el trabajo que realizará el hilo [3].

```
let printNumbers (max: obj) =
    for i in 1 .. (max :?> int) do
        printfn "%d" i

ThreadPool.QueueUserWorkItem(new WaitCallback(printNumbers), box 5)
```

Este ejemplo muestra cómo se recupera un hilo del conjunto disponible en `ThreadPool`. El método `QueueUserWorkItem` recibe como parámetro a una nueva instancia de `WaitCallback` la cual toma a la función `printNumbers` como parámetro y a un objeto tipo `obj` para uso dentro de la función que se ejecutará.

2.2.2 `BackgroundWorkers`

.NET ofrece otra solución para el uso de hilos a través de la clase `System.ComponentModel.BackgroundWorker`. Esta clase corre en su propio hilo de sistema operativo, cuenta con múltiples métodos de ejecución y variables mutables para el almacenamiento de resultados. A continuación se muestra un ejemplo de cómo se utiliza `BackgroundWorker`, desde su creación hasta la recuperación de su resultado [4].

```
let w = new BackgroundWorker()
w.DoWork.Add(fun args ->
    let mutable count = 0
    for i in 1 .. 5 do
        count <- count + 1
    args.Result <- box count)

w.RunWorkerCompleted.Add(fun args ->
    MessageBox.Show(sprintf "Result = %A" args.Result) |> ignore)

w.RunWorkerAsync()
```

`BackgroundWorker` ejecuta la función que recibe `DoWork.Add`; en este ejemplo, la función itera del uno al cinco, incrementando la variable `count` en cada iteración y almacenando el resultado en la variable `args.Result` al final. En cuanto termine su ejecución, se manda a llamar la función que se dio de alta en la llamada a `RunWorkerCompleted.Add`, la cual crea una ventana que muestra el resultado. El objeto se manda a llamar con la función `RunWorkerAsync`.

2.3 Las desventajas de los hilos

Las desventajas de utilizar hilos directamente no se limitan al costo de tiempo y recursos que puede implicar. El uso de hilos incluye memoria compartida, y esto introduce problemas de condición de carrera. El uso

de candados puede solucionar el problema anterior, pero a su vez introduce algo igual o peor: *deadlocks*. Finalmente, el uso de candados puede eliminar toda mejora obtenida al paralelizar un programa ya que termina serializando el acceso a recursos compartidos [3].

Aunque el uso directo de hilos está perfectamente permitido, F# ofrece opciones más abstractas para facilitar el uso de los mismos. A través de patrones de diseño y clases que ocultan el funcionamiento de bajo nivel, el usuario puede implementar el mismo paralelismo o concurrencia en su programa sin mayor esfuerzo.

3 Patrones de diseño para el paralelismo y concurrencia

3.1 Asynchronous Programming Model (APM)

Historicamente, APM ha sido el modelo preferido para lograr paralelizar programas desarrollados en .NET, sin embargo, puede llegar a introducir complejidades innecesarias al implementarse en F#. Este modelo intenta dividir una tarea asíncrona en dos partes principales, una que se ejecuta al inicio y otra al fin. Las operaciones que se mandan a llamar al inicio llegan el prefijo de `Begin` y aquellas que se mandan a llamar al fin llevan el prefijo de `End`. Finalmente, las transiciones entre métodos se coordinan y pasan resultados a través de la interface `IAsyncResult` [3].

APM abstrae el manejo de hilos para el usuario, pero al utilizarse introduce un nuevo conjunto de problemas que complica el flujo del código. El siguiente ejemplo se encuentra en [3] y se incluye para demostrar lo complejo que puede hacerse el uso de este modelo.

```
let processFileAsync (filePath : string) (processBytes : byte[] -> byte[]) =

    let asyncWriteCallback =
        new AsyncCallback(fun (iar : IAsyncResult) ->
            let writeStream = iar.AsyncState :?> FileStream

            let bytesWritten = writeStream.EndWrite(iar)
            writeStream.Close()

            printfn
                "Finished processing file [%s]"
                (Path.GetFileName(writeStream.Name))
        )

    let asyncReadCallback =
        new AsyncCallback(fun (iar : IAsyncResult) ->
            let readStream, data = iar.AsyncState :?> (FileStream * byte[])

            let bytesRead = readStream.EndRead(iar)
            readStream.Close()

            printfn
                "Processing file [%s], read [%d] bytes"
                (Path.GetFileName(readStream.Name))
                bytesRead

            let updatedBytes = processBytes data

            let resultFile = new FileStream(readStream.Name + ".result",
                                            FileMode.Create)
```

```
        let _ =
            resultFile.BeginWrite(
                updatedBytes,
                0, updatedBytes.Length,
                asyncWriteCallback,
                resultFile)

        ()
    )

let fileStream = new FileStream(filePath, FileMode.Open, FileAccess.Read,
                                FileShare.Read, 2048,
                                FileOptions.Asynchronous)

let fileLength = int fileStream.Length
let buffer = Array.zeroCreate fileLength

let state = (fileStream, buffer)

printfn "Processing file [%s]" (Path.GetFileName(filePath))
let _ = fileStream.BeginRead(buffer, 0, buffer.Length, asyncReadCallback, state)
()
```

Entre los problemas que introduce se encuentra el rastreo del flujo al mandar a llamar múltiples tareas asíncronas, excepciones a tiempo de ejecución si es que la conversión de información recuperada desde la interface `IAsyncResult` no se hace correctamente, y problemas con el manejo de memoria si es que las llamadas de terminación (operaciones con el prefijo *End*) no se mandan a llamar [3]. Afortunadamente los siguientes modelos intentan evadir estos problemas, ofreciendo una mejor manera de paralelizar el código serial.

3.2 Asynchronous Workflows

Los flujos asíncronos que proporciona F# permiten realizar operaciones asíncronas sin la necesidad de llamadas de retorno (*callbacks*) explícitas. Se puede escribir código como si fuera una ejecución síncrona, pero en realidad, el código se ejecutará asíncronamente, suspendiendo y resumiendo las operaciones como operaciones asíncronas completas.

3.2.1 Las bibliotecas Async

El secreto detrás de los flujos de trabajo asíncronos es que el código está envuelto en un bloque `async` y no es ejecutado inmediatamente. En lugar de eso, la operación que el código realiza es devuelta en forma de un objeto de tipo `Async<'T>`, el cual se puede pensar como una operación asíncrona que eventualmente regresa una instancia de `'T`. Como el tipo `'T` será extraido del objeto dependerá del módulo `Async` y del constructor de expresiones computacionales `async`. Cada que un `let!`, `do!`, o cualquier acción similar sea realizada, el constructor de expresiones computacinoales `async` empezará la tarea asíncronamente y se ejecutará el resto de la operación una vez que esa tarea se complete.

Existen varios métodos disponibles para comenzar un flujo de trabajo asíncrono. El más simple es invocar `Async.Start`, el cual toma como parámetro un `Async<unit>` y simplemente comienza ejecutándolo asíncronamente. Si se quiere que la tarea asíncrona regrese un valor, se necesita esperar a que se complete la operación llamando `Async.RunSynchronously`. El siguiente ejemplo define una función `getHtml` que recibe una URL como parámetro y regresa el contenido de la página. Esta función regresa un tipo `Async<string>`.

```
open System.IO
open System.Net
open System.Microsoft.FSharp.Control.WebExtensions

let getHtml (url : string) =
    async {

        let req = WebRequest.Create(url)
        let! rsp = req.AsyncGetResponse()

        use stream = rsp.GetResponseStream()
        use reader = new StreamReader(stream)

        return! reader.AsyncReadToEnd()
    }

let html =
    getHtml "http://en.wikipedia.org/wiki/F_Sharp_programming_language"
    |> Async.RunSynchronously
```

`Async.RunSynchronously` no es útil por sí solo porque bloquea el thread esperando a que la operación termine. Usualmente este método se llama inmediatamente despues de una llamada `Async.Parallel`, la cual toma como parámetro un `seq<Async<'T>>` y comienza todas las secuencias en paralelo. El resultado combinado es una instancia de `Async<'T[]>`. El siguiente código aplica la funcion `getHtml` a una serie de páginas web en paralelo.

```
let webPages : string[] =
    [ "http://www.google.com"; "http://www.bing.com"; "http://www.yahoo.com" ]
    |> List.map getHtml
    |> Async.Parallel
    |> Async.RunSynchronously
```

Otro ejemplo de uso de las bibliotecas Async es calcular una serie de Fibonacci en paralelo. El siguiente código define una función recursiva para calcular el siguiente número en la serie de Fibonacci, la cual es aplicada a un arreglo a través de `async`

```
let rec fib x = if x <= 2 then 1 else fib(x-1) + fib(x-2)

let fibs =
    Async.Parallel [ for i in 0..40 -> async { return fib(i) } ]
    |> Async.RunSynchronously
```

3.2.2 Ventajas y desventajas de Asynchronous Workflows

Una de las ventajas de utilizar flujos de trabajo asíncronos en F# es que se hace muy sencillo el manejo de excepciones y soporte de cancelación, algo que es muy difícil cuando se utiliza APM.

Los flujos de trabajo asíncronos son buenos para realizar operaciones de entrada/salida en paralelo. Debido a que la biblioteca es una simple envoltura encima del pool the threads, usarla no garantiza que vas a mejorar el desempeño. Cuando se ejecuta código en paralelo, se debe de tomar en cuenta el número de procesadores por núcleo, la coherencia en la memoria caché y la carga existente en el CPU. Mientras que los flujos de trabajo asíncronos de F# hacen muy fácil realizar muchas operaciones al mismo tiempo, no hay un límite de subprocesos que se ejecutan para asegurar un uso óptimo. Para realizar paralelismo a nivel de CPU, se debería de considerar utilizar la extensión paralela de .NET.

3.3 Programación paralela

La programación paralela consiste en dividir una operación en n partes para obtener una velocidad de procesamiento n veces mayor. La forma más fácil de realizar programas paralelos en .NET es a través de la Extensión Paralela de la plataforma .NET (PFX). Utilizando el PFX no hay necesidad de controlar manualmente los threads y el pool de threads[1].

3.3.1 `Parallel.For`

El primer paso que se tiene que realizar para paralelizar aplicaciones es cambiar los ciclos `for` por `Parallel.For` o `Parallel.ForEach` dentro del espacio de nombres `System.Threading`. Hay que recordar que introducir ciclos paralelos puede generar errores cuando los cálculos realizados dependen de una iteración anterior. El siguiente ejemplo multiplica dos matrices y regresa una matriz resultante.

```
open System
open System.Threading.Tasks

/// Multiplicación de matrices utilizando PFX
let matrixMultiply (a : float[,]) (b : float[,]) =

    let aRow, aCol = Array2D.length1 a, Array2D.length2 a
    let bRow, bCol = Array2D.length1 b, Array2D.length2 b
    if aCol <> bRow then failwith "Array dimension mismatch."

    // Abrir espacio para la matriz resultante, c
    let c = Array2D.create aCol bRow 0.0
    let cRow, cCol = aCol, bRow

    // Calcular cada fila de la matriz resultante
    let rowTask rowIdx =
        for colIdx = 0 to cCol - 1 do
            for x = 0 to aRow - 1 do
                c.[colIdx, rowIdx] <-
                    c.[colIdx, rowIdx] + a.[x, colIdx] * b.[rowIdx, x]
        ()

    let _ = Parallel.For(0, cRow, new Action<int>(rowTask))

    // regresar la matriz resultante
    c
```

Construido encima de PFX se encuentra el módulo `Array.Parallel`, que contiene algunos métodos del módulo `Array`, como `map`, `mapi` y `partition`, la única diferencia es que estos métodos completan las operaciones de forma paralela.

La estructura fuente dentro del paralelismo de PFX es el objeto `Task`, similar a `Async<'T>`, que representa el cuerpo de cierto trabajo que será completado después. Nuevas tareas pueden ser creadas utilizando uno de los métodos sobreescritos de `Task.Factory.StartNew`. Una vez creada, la tarea puede ser agendada para ser ejecutada en paralelo, aunque la biblioteca de PFX determinará cuantas tareas se crearán en algún momento, dependiendo de los recursos disponibles. Para recuperar el valor de una tarea, sólo es necesario acceder a su propiedad `Result`, el cual puede almacenar el resultado de una tarea ya terminada, esperar a que la tarea termine si es que está en ejecución o comenzar la tarea si el hilo actual no ha empezado su ejecución. Además es posible combinar múltiples tareas con las primitivas síncronas `Task.WaitAll` y `Task.WaitAny`.

Otro beneficio de las tareas es que manualmente se puede cancelar a través de mecanismos de flujo de trabajo asíncrono.

PFX introduce nuevas colecciones para resolver el problema de estructuras de datos no concurrentes. El espacio de nombres `System.Collections.Concurrent` contiene los tipos de colecciones estándar que se esperan, excepto que pueden ser compartidos libremente entre los hilos de ejecución. Algunas colecciones dentro de este espacio de nombres son `ConcurrentQueue`, `ConcurrentDictionary` y `ConcurrentBag` (que es equiparable al `HashSet<_>`).

4 Conclusiones

F# ofrece varias opciones para paralelizar programas secuenciales, desde el manejo directo con hilos a nivel sistema operativo hasta modelos y patrones de diseño que abstraen el manejo de bajo nivel. Además, la infraestructura de .NET incluye muchas clases útiles para la concurrencia y tareas asíncronas, aunque no todas ofrecen la misma simplicidad en F# que ofrecen otros lenguajes de .NET. Al ofrecer opciones de diferente grado de control y complejidad, F# hace un buen trabajo al atacar los temas de paralelización, concurrencia y tareas asíncronas, aunque todavía hay campo para mejorar; la existencia de expresiones computacionales, tipos inmutables y la inclusión del paradigma funcional en su sintaxis son una ventaja mientras que la disponibilidad y funcionamiento de varios tipos de candados para el manejo de memoria compartida puede mejorar. En general, F# cumple con las características necesarias para mejorar la programación serial a través del diseño paralelo y concurrente.

Notas

[1] El ambiente paralelo sólo existe en versiones del CLR 4.0. Si se crean aplicaciones de `F#` en ambientes .NET 2.0, 3.0 ó 3.5, no se podrá tomar ventaga de todas las bibliotecas PFX. Sin embargo, las bibliotecas de flujo de trabajo asíncrono se encuentran soportadas en las versiones anteriores de .NET.

Referencias

[1] Microsoft. *F# at Microsoft Research.*
http://research.microsoft.com/en-us/um/cambridge/projects/fsharp/ Accedido el 25 de octubre del 2012

[2] MSDN. *BackgroundWorker Class*
http://msdn.microsoft.com/en-us/library/system.componentmodel.backgroundworker(v=vs.100).aspx#Y0

[3] Smith, C. *Programming F#.* Sebastopol: O'Reilly Media, Inc., 2009

[4] Syme, D., Granicz, A., & Cisternino, A. *Expert F# 2.0.* New York: Apress, 2010

Go, El lenguaje de programación de Google

Thania Guadalupe Cerecedo Zamarripa (A01160864)
Instituto Tecnológico y de Estudios Superiores de Monterrey
Campus Estado de México
Atizapán de Zaragoza, Estado de México, México.

31 de octubre, 2012.

Resumen

Este artículo describe como el lenguaje de programación concurrente Go esta diseñado para cumplir con el desafío de la programación multinúcleo y para hacer la programación paralela más fácil.

1 Introducción

En el año 2009 Google Inc. anunció un nuevo lenguaje de programación llamado Go, que es un lenguaje de programación concurrente y compilado inspirado en la sintaxis de C. Go está diseñado para incrementar la eficiencia, para que así pueda ser usado para escribir grandes aplicaciones con el menor tiempo de compilación. Soporta concurrencia usando *Goroutines* y un canal de comunicación tipo CPS, y gracias a ello, hace más fácil el escribir programas para obtener el máximo rendimiento de las maquinas multinúcleo y en red[1]. Los ingenieros que desarrollan el lenguaje, lo describen como rápido, divertido y productivo, donde pueden escribir sus programas más rápido, más efectivo y que soporta los grandes sistemas distribuidos que conectan miles de maquinas y el tipo de problemas que se encuentran al escribir ese tipo de programas.

2 Desarrollo

2.1 Soporte de Go para la concurrencia

Una distinción muy importante entre paralelismo y concurrencia es que el paralelismo, consiste en ejecutar varias cosas simultáneamente y concurrencia es una forma de controlar las cosas que se ejecutan de forma simultánea. Se puede decir que la concurrencia es la coordinación de computaciones hechas en paralelo, y Go provee rutinas que permiten ejecutarlas y crear paralelismo, además de crear canales que permiten controlar estas instrucciones en paralelo por medio de comunicación explicita[2].

La manera en que Go hace posible utilizar múltiples núcleos, es dejando proponer al tiempo de ejecución, cuantos threads del sistema operativo usar para ejecutar las goroutines, y luego mezclar esas rutinas entre esos threads.

2.2 Goroutines

Las Gourutines son funciones ejecutadas en un thread separado. Para inicializarlo, se utiliza el prefijo go en la función llamada.

```
go count(name, URL)
```

Esta declaración arrancará la función count como una goroutine en un thread separado. Esto hace una llamada asíncrona y el control no esperará a que termine la ejecución de count antes de ejecutar la siguiente declaración, y cuando la goroutine termine, saldrá silenciosamente. Las gourutines comparten la misma memoria que las demás y del thread principal de ejecución

Múltiples goroutines pueden ser ejecutadas en el mismo sistema de threads.

Por default en el tiempo de ejecución de Go, sólo se usará un procesador para calendarizar las goroutines, y para usar más de un procesador, se utiliza la función runtime.GOMAXPROCS.

Por ejemplo, si se quieren utilizar 4 procesadores, la instrucción es:

```
import ("runtime")

func main() {
 runtime.GOMAXPROCS(4)
}
```

2.3 Canales

Los canales son la mayor forma de sincronización de Go. Pueden ser usadas para enviar y recibir valores entre goroutines, y se utilizan de la siguiente manera:

```
1  ch := make(chan int)

2  go func() {
3  c:= <-ch
4  }()
5  ch <- 99
```

En la línea 1, se crea un nuevo canal usando make. Los canales por default son sacados del buffer y se bloquearán al enviar y recibir. Después se genera una nueva goroutine que recibirá un valor por medio del canal (Línea 3). Finalmente, se envía el número 99 través del canal (Línea 5).

Para enviar un valor a través del canal, se utiliza el operador ¡- con el canal en el lado izquierdo (Línea 5), y Para recibir un valor se utiliza el canal en el lado derecho del operador ¡-.

El orden para enviar y recibir es importante, ya que si se tuviera el canal ch¡-99 antes de la línea 2, el programa se bloquearía y nunca ejecutaría la declaración go, mientras que un canal sin memoria intermedia bloquearía el send y el receive.[3]

2.4 Waitgroup

Los waitgroups son una mejor manera de sincronizar la compleción de los goroutines, y están presentes en el sync package. Se puede re escribir el código anterior utilizando waitgroups.

```
1 var wg sync.WaitGroup

2 for i:=0; i<n; i++ {
3  wg.Add(1)
4  go func() {
5
6  wg.Done()
7  }()
8 }
9 wg.Wait()
10}
```

Se puede ver como el goroutine principal llama a Add para fijar el número de goroutines para esperar (Línea 3). Cuando cada goroutine termina de ejecutar, se llama el método Done (Línea 6) en el waitgroup, después la rutina principal espera a que terminen todos los goroutines hijos llamando a Wait (Linea 9).

2.5 Select

La declaración select es usada para escoger entre un send y un receive de entre un grupo de canales. La estructura de la declaración es parecida a la de un switch, con cada caso siendo el send o el receive de un canal. Cada uno de estos casos son evaluados de arriba hacia abajo, y al final uno es seleccionado para ser ejecutado, de entre todos los que pueden proceder.

2.6 Locks

En la paquetería de sync, hay dos tipos de locks: Mutex y Read Writer Lock, utilizados para construir un nivel más alto de mecanismos de sincronización .

2.7 Once

La estructura Once puede ser usada para ejecutar una función en particular una sola vez. Por ejemplo:

```
1 var once sync.Once

2 for i:=0; i<n; i++ {

3  go func() {
4
5    once.Do(cleanup)
6  }()
7 }
```

En este código, aunque varias goroutines alcanzarán la línea 5, sólo una de ellas ejecutará la función cleanup.

2.8 Paralelización

La aplicación de paralelizar cálculos entre múltiples núcleos de CPU, permite separar por piezas el cálculo para que puedan ser ejecutadas independientemente. Puede ser paralelizada con un canal, y manda una señal cuando cada pieza se complete.

Por ejemplo si se tiene una operación que resulta costosa para calcular cierto número de vectores y el valor de la operación en cada sección es independiente, es ideal aplicar:

```
1 type Vector []float64
2 func (v Vector) DoSome(i, n int, u Vector, c chan int) {
3    for ; i < n; i++ {
4        v[i] += u.Op(v[i])
5    }
6    c <- 1
7 }
```

En la línea 2 se aplica la operación desde v[i], v[i+1] ... hasta v[n-1].

Se lanzan las piezas independientes en un ciclo, una por CPU. Se pueden completar en cualquier orden y solo se cuentan las señales de los procesos completos drenando el canal después de lanzar todos los goroutines[4].

```
1 const NCPU = 4  // n\'umero de n\'ucleos del CPU

2 func (v Vector) DoAll(u Vector) {
3    c := make(chan int, NCPU)
4    for i := 0; i < NCPU; i++ {
5        go v.DoSome(i*len(v)/NCPU, (i+1)*len(v)/NCPU, u, c)
6    }
    // Drenar el canal
7    for i := 0; i < NCPU; i++ {
8        <-c    // espera hasta que se complete una tarea
9    }
10}
```

3 Conclusiones

El lenguaje de programación Go de Google es relativamente nuevo, con tan sólo 2 o 3 años desde su lanzamiento, y aunque sigue en una fase experimental, los ingenieros de Google han probado su velocidad en Web Crawl contra lenguajes como Python, Ruby y Scala. Este lenguaje ha recibido muy buenas criticas entre comunidades de programadores, que usándolo por poco tiempo, se han adaptado muy bien y lo han descrito como efectivo y rápido, aunque nunca hayan programado en un ambiente paralelo.

El lenguaje es fácil de instalar, incluye muchas bibliotecas y tiene la documentación suficiente para que la gente pueda empezar a usarlo y esté a la par de lenguajes como Erlang. Go recopila aspectos de C++ y C, y escribir en este lenguaje tiene muchas ventajas, pero es importante comprender que tiene sus propias propiedades y convenciones[5].

Referencias

[1] The Go Programming Language. http://golang.org/ Accedido el 3 de octubre del 2012.

[2] Go Team. The Go programming language specification. Technical Report. http://golang.org/doc/doc/go spec.html Accedido el 30 de octubre del 2012.

[3] Multi-Core Parallel Programming in Go. http://www.ualr.edu/pxtang/papers/ACC10.pdf

[4] Effective Go. http://golang.org/doc/effective$_go.htmlconcurrency.Accedidoel31deoctubredel2012.$

[5] Programming in Go: Creating Applications for the 21st Century. Mark Summerfield, 2nd Edition. Addison-Wesley Professional, 2012.

Capacidades concurrentes del lenguaje Io

Gerardo Galíndez Barreda (A01164096) Juan Ramón Fernández Álvarez (A01164922)
Instituto Tecnológico y de Estudios Superiores de Monterrey
Campus Estado de México
Atizapán de Zaragoza, Estado de México, México.

3 de octubre, 2012.

Resumen

Este documento describe el funcionamiento básico del lenguaje Io, un lenguaje de programación orientado a prototipos con facilidades importantes de concurrencia y de metaprogramación.

1 Introducción a Io

Io es un lenguaje de programación orientado a objetos basado en prototipos, dinámico y fuertemente tipado. Su autor, Steve Dekorte, cita a Smalltalk, Newtonscript, Lua y Lisp como sus principales influencias. Al igual que Ruby, Io es un lenguaje de programación con facilidades importantes de metaprogramación que le permiten modificar incluso la sintaxis del lenguaje.

Además, al igual que Erlang, Scala y Clojure tiene un modelo de concurrencia orientado a actores, en el cuál un componente corre en su propio *thread*, aislado del resto.

1.1 Programación orientada a prototipos

Io sigue el paradigma de programación orientada a objetos basada en prototipos, al igual que Javascript o Self. La programación basada en prototipos es un concepto similar a la programación orientada a objetos, sin embargo la unidad funcional de un prototipo es el objeto en sí, en lugar de una clase.

En la programación en prototipos, a diferencia de la programación orientada a objetos, no se definen clases con métodos y atributos. Se define un objeto base, que es genérico y se le asignan métodos y atributos. Posteriormente, para instanciar un nuevo objeto, se clona el objeto existente.

Debido a sus características, la programación basada en prototipos tiene ciertas ventajas con respecto a su contraparte.

- El modelo de programación favorece el paso de mensajes entre instancias.
- Las características de una clase se pueden modificar a tiempo de ejecución al estilo de Ruby de forma nativa, ya que el prototipo en sí se construye a tiempo de ejecución.
- En general, es más sencillo hacer un diseño flexible.
- Delegar acciones es sencillo, ya que se pueden pasar mensajes entre prototipos

Estas cuatro características hacen que los prototipos sean buenos candidatos para formar lenguajes concurrentes. Al igual que Erlang, Clojure u otros lenguajes funcionales orientados a la concurrencia, el paso de mensajes es una parte importante de estos lenguajes.

La implementación práctica que mejor demuestra las capacidades de un lenguaje basado en prototipos para la concurrencia, probablemente sea Node.js, el cuál es una implementación de Javascript diseñada principalmente para ofrecer soporte de entrada/salida concurrente sin bloqueos.

1.2 Descripción general de Io

Io es un lenguaje muy sencillo, su sintaxis y funcionamiento general se puede explicar muy brevemente por lo que en esta sección se describirá su uso general y en la siguiente, demostraremos con un ejemplo sencillo cómo se puede modelar un objeto sencillo. Esta sección no es una referencia a profundidad de Io, sino que simplemente explica las bases para que puedan interpretarse los ejemplos de concurrencia con claridad.

En Io se crean objetos clonando otros objetos, los objetos son colecciones de *slots*. Puede verse como una tabla de hash. Primero que nada, en Io se asignan objetos a los *slots* usando los operadores **=**, **:=** y **::=**. Para usar un *slot* de un objeto, se le pasa un mensaje.

```
Io> Example := Object clone
```

Io tiene soporte mínimo para colecciones, se pueden crear listas y mapas con sus características normales, presentes en otros lenguajes de programación. Para crear una lista o un mapa se clonan los objetos List o Map, correspondientemente.

Al igual que Ruby, se pueden implementar diferentes operadores que extiendan la gramática del lenguaje. A diferencia de Ruby, casi cualquier cosa se puede convertir en un operador. Para saber agregar, modificar, eliminar o saber qué operadores reconoce Io, se puede usar el objeto **OperatorTable**.

Los mensajes se envían especificando un objeto seguido del mensaje, separado por un espacio en blanco. Los mensajes son síncronos (a menos que se pida explícitamente lo contario) y está garantizado de que el objeto los va a recibir.

El último punto importante de Io es que tiene capacidades de *reflection* (reflexión). Hay dos tipos de reflexión, con los objetos y con los mensajes. Ambos tipos de reflexión tienen que ver con los *slots* de un objeto determinado de Io. Como un prototipo se puede modificar en todo momento, la reflexión en Io está presente en todo momento.

2 Ejemplos de Io

Para poder comprender mejor el como funciona Io lo mejor que podemos hacer es escribir un pequeño programa. Empezaremos creando un objeto.

```
Io> Animal := Object clone
==> Animal_0x1f4d3b8:
    type    = "Animal"
Io> Animal whatis = "Un ser vivo"
    Exception: Slot whatis not found. Must define using := operator before updating.
Io> Animal whatis := "Un ser vivo"
==> A living being of sorts
Io> Animal whatis
==> A living being of sorts
```

Lo primero que hicimos fue crear un objeto de tipo Animal, clonándolo de Object. Después intentamos asignar un slot al objeto Animal, pero utilizando el operador de asignación. Como podemos ver, la consola nos dice que *whatis* no se encuentra definido, por lo que no podemos hacer una asignación y hay que utilizar el operador para definir si es que eso es lo que deseamos. En la tercera línea creamos el slot *whatis* y en la cuarta verificamos que funciona. Ahora jugaremos con un poco de herencia.

```
Io> Badger := Animal clone
==> Badger_0x1e9e528:
    type    = "Badger"
Io> Badger whatis
==> A living being of sorts
Io> Badger whatis = "A dancing mammal associated with mushrooms and snakes"
==> A dancing mammal associated with mushrooms and snakes
Io> Badger whatis
==> A dancing mammal associated with mushrooms and snakes
Io> Animal whatis
==> A living being of sorts
```

Empezamos creando un objeto tipo Badger a partir de Animal. Como podemos ver en la segunda instrucción, desde el momento de su creación Badger ya comparte el slot *whatis* de Animal. En la siguiente instrucción lo que hacemos es escribirle a Badger su propio slot *whatis*. Hay que hacer notar que a diferencia del método *animal* solamente asignamos *whatis* en vez de definirlo. En las últimas líneas ya vemos como Badger tiene su propio *whatis* y Animal sigue conservando el suyo. Antes de pasar a algo más complicado le añadiremos métodos a ambos.

```
Io> Animal hello := method ("Hello from Animal" println)
==> method(
    "Hello from Animal"
)
Io> Animal hello
Hello from Animal
==> Hello from Animal
Io> Badger hello
Hello from Animal
==> Hello from Animal
Io> Badger hello = method("MUSHROOMS!" println=
==> method(
    "MUSHROOMS!"
)
Io> Animal hello
Hello from Animal
==> Hello from Animal
Io> Badger hello
==> MUSHROOMS!
MUSHROOMS!
```

Algo que podemos ver es que method se comporta como si fuera un objeto. Esto se debe a que en Io method es un objeto, por lo que podemos asignarlo a un slot cualquiera.

3 Modelo de concurrencia

El autor de Io, Steve Dekorte, le mencionó en una entrevista a Bruce Tate que uno de los objetivos principales de Io era tener una sintaxis muy sencilla y consistente, pero que fuera muy flexible. Io es en lenguaje mucho más lento que otros lenguajes de scripting, sin embargo, al estar escrito en C, Steve Dekorte creo una interfaz con *SIMD* (Single Instruction, Multiple Data), la cuál permite a Io tener capacidades buenas de concurrencia.

Io usa un calendarizador simple *FIFO* (First-In, First-Out. Primero en entrar, primero en salir), la primer tarea que entra es la primera en salir. Esto es muy diferente a otros tipos de lenguajes en lo que se usa un calendarizador multitarea, apropiativo, en el que el calendarizador tiene completo control sobre la ejecución

de un programa. Al usarse con un lenguaje en el que hay *side effects* (efectos secundarios) el flujo y el efecto de un programa se vuelve no determinista.

Debido a que en Io los objetos son mutables, tienen *side effects* y por sus características dinámicas, el calendarizador es *FIFO*, lo que hace que el flujo y efecto de un programa sean deterministas. En la descripción posterior se hace un análisis más a detalle de tales estructuras.

Io cuenta con tres componentes principales de concurrencia: *coroutines*, *actors* y *futures*. Los tres componentes ofrecen distintos niveles de control y deben de ser usados de acuerdo al problema que esté resolviendo. En esta sección se describen los componentes de concurrencia de Io y sus capacidades. En la siguiente sección se presentan ejemplos de su uso.

3.1 Corrutinas (*coroutines*)

Una corrrutina es la unidad fundamental de concurrencia en Io, como lo son los objetos Thread de Java, los pthreads de C o los procesos de Erlang. Las corrutinas consisten en mecanísimos simples para comenzar o suspender la ejecución de un bloque de código. En sí, son simplemente funciones con múltiples puntos de regreso, para continuar el flujo de ejecución o para suspenderlo.

Al igual que lenguajes como Erlang, y a diferencia de Java, los threads creados por Io no son nativos o de nivel de sistema operativo, sino que son específicos de Io. Esto se hace con el mismo objetivo que en Erlang, evitar el alto consumo que representan los threads nativos y simplificar su uso mediante abstracciones de alto nivel.

Las corrutinas son asíncronas. Al igual que el resto de Io, las corrutinas tienen una sintaxis muy simple. Los operadores para iniciar corrutinas son **@** y **@@**.

- **@**. Inicia la corrutina y regresa un *future*
- **@@**. Inicia la corrutina y regresa un *nil*. Inicia en su propio thread.

3.2 Actors

Un *actor* es un objeto que vive en su propia corrutina en la cuál procesa sus mensajes asíncronos en una forma similar a la de Erlang. En Io no existe un concepto de *mailbox*, al llamar a cualquier método se crea un mensaje, por lo que estos mensajes están implícitos en los objetos.

Cualquier objeto que recibe un mensaje asíncrono, se convierte automáticamente en un actor. Para enviar un mensaje asíncrono a un objeto se usa la misma sintaxis que para las corrutinas. Una vez que el objeto recibe dicho mensaje, incializa de forma automática (si todavía no existe) una cola de mensajes.

3.3 Futures

Los *futures* obtienen su nombre de su comportamiento. Son objetos que serán el resultado de una llama asíncrona. Un envío de un mensaje asíncrono, regresa un *future* que una vez que la llamada haya terminado, contendrá el resultado. El objetivo de esto es simplificar los bloqueos, candados o algún otro mecanismo de sincronización concurrente.

Una vez que se tiene un *future*, se puede usar. En caso de que el resultado aún no esté terminado, la corrutina del objeto que contiene al *future* se bloquea y espera a que la llamada termine.

Aquí es donde realmente sale a relucir el modelo de concurrencia de Io y su extraña decisión por el calendarizador *FIFO*. El hecho de que las llamadas sean deterministas en lugar de no deterministas hacen que los futures puedan tener detección automática de deadlocks.

Esto hace que los futures sean efectivamente una gran opción para programar acciones concurrentes como callbacks que reciben llamadas asíncronas. Además, la detección automática de deadlocks hace que sea más sencillo optimizar y depurar el código de una aplicación.

4 Ejemplo de Concurrencia

Ya que hemos explicado las bases de concurrencia, creemos que lo mejor sería demostrar como funciona a través de un ejemplo. Comenzaremos con un ejemplo sencillo.

```
Delorean := Object clone
Delorean year := 0
Delorean now := method(
    "Current Year: " println;
    Delorean today println
)
Delorean run := method(
    for(i, 1, 731337,
        Delorean year = 0;
        Delorean year = Delorean year + i
    )
)

Delorean today := Delorean @run
"Back to the future!" println
Delorean now

"" println

Delorean today = Delorean run
"Now in slooooow mo..." println
Delorean now
```

En este ejemplo tenemos un objeto Delorean. Si leemos el código vemos que lo que hace es calcular dos veces "today" y posteriormente, imprimir el resultado tras hacer unas impresiones. Si nosotros corremos el programa notaremos un comportamiento un tanto peculiar. La primera vez que lo calcula imprime los mensajes inmediatamente, y la segunda ocasión no imprime nada hasta después de un rato. Esto se debe a que en la primera ocasión hicimos uso de un *future*, es decir Io hizo un nuevo thread en el que calculaba *run* mientras que seguía ejecutando código hasta que el valor de run fue necesario, donde hace una pausa hasta que obtiene su valor. Por otro lado la segunda ocasión las impresiones no salen hasta que termina de calcular run ya que esta corriendo sobre el mismo proceso. Ahora por tradición hicimos otro programa en el que calculamos Pi haciendo uso de futuros.

```
Pi := Object clone
Pi num_rects := 100000
Pi width := method( 1 / Pi num_rects)
Pi area := method( Pi width * (Pi sum_a  + Pi sum_b))

Parta := Object clone
Parta count := 0
Parta mid := method(i, (i +0.5 ) * Pi width)
Parta height := method(i, 4.0 / (1.0 + Parta mid(i) * Parta mid(i)))
Parta sum := method(
```

```
    Parta count = 0;
    for(i, 1, Pi num_rects / 2, Parta count = Parta count + Parta height(i))
)

Partb := Object clone
Partb count := 0
Partb mid := method(i, (i +0.5 ) * Pi width)
Partb height := method(i, 4.0 / (1.0 + Partb mid(i) *Partb mid(i)))
Partb sum := method(
    Partb count = 0;
    for(i,1 + Pi num_rects  / 2, Pi num_rects, Partb count = Partb count + Partb height(i))
)

"Start" println
Pi sum_a := Parta @sum
Pi sum_b := Partb @sum
"Processing..." println
Pi area println
```

Al igual que con el programa anterior, cuando nosotros corremos el programa notamos que obtenemos el mensaje de "Start" y "Processing..." de inmediato; de nuevo, esto se debe a que estamos haciendo uso de futuros. Si nosotros no hubieramos usado el futuro, es decir simplemente quitando @ antes de sum, el mensaje de "Processing..." se habría impreso hasta después de haber terminado de procesar las sumas. Esto demuestra lo sencillo que es usar la concurrencia en Io.

5 Comparación de Io con otros lenguajes

A través del artículo hemos comparado a Io con otros lenguajes de programación para ejemplificar mejor su funcionamiento. En esta sección presentamos un resumen de las comparaciones hechas en las secciones pasadas.

En casos generales, el rendimiento de Io es bastante inferior a los lenguajes más veloces como C y C++, sin embargo esto depende mucho de qué problema se esté resolviendo. Para casos seriales, el rendimiento de Io es inferior para la vasta mayoría de las situaciones. Sin embargo, para casos paralelos, en problemas altamente concurrentes, Io puede ser incluso más veloz que C. A mayor concurrencia y mayores recursos distribuidos, mayor la capacidad de Io. Esto deja atrás incluso a lenguajes como Erlang, sin embargo, Io no cuenta con las mismas capacidades de escalabilidad.

Io es un lenguaje muy pequeño, por lo que su *footprint* de memoria también es bastante pequeño. En general, consume menos que casi cualquier otro lenguaje que corra en una máquina virtual. De los lenguajes mencionados, solamente C y C++ son más pequeños que Io en términos de consumo de memoria.

5.1 Comparación por lenguajes

- **Java**. La diferencia principal entre Io y Java es el manejo de threads. Como se mencionó anteriormente, Java usa threads nativos a la máquina virtual, por lo que cada objeto de la clase Thread es muy costoso. En Io no existe como tal un objeto de tipo Thread, son reemplazados por las corrutinas las cuáles son bloques de código con múltiples salidas que corren en su propio thread. Tales threads son nativos a Io, no al sistema operativo. El calendarizador puede o no ser *preemptive* (apropiativo), depende de la implementación de la máquina virtual.

- **C/C++**. Comparando a Io con C o C++ como lenguajes de programación, son prácticamente opuestos. Io es un lenguaje basado en prototipos, dinámico e interpretado con sintaxis minimalista y por

otra parte C es un lenguaje estructurado, C++ es orientado a objetos, ambos compilados, estáticos y con una sintaxis compleja. En términos de concurrencia, C y C++ usan threads nativos al sistema operativo. Además, los calendarizadores son *preemptive* (Apropiativos) a diferencia de IO que es *FIFO*

- **Erlang**. Erlang y Io comparten el concepto de actores junto con Scala (a menor proporción) en el que sentido de que cada actor tiene su propio espacio de ejecución, no pueden comunicarse con otros actores (o procesos en el caso de Erlang) sin usar mensajes asíncronos y cada uno es efectivamente concurrente. El calendarizador de Erlang es muchísimo más complejo y avanzado. Io es mucho más minimalista, lo que hace que escribir una aplicación altamente concurrente sea más sencillo que en Erlang, al costo de la escalabilidad. Cada proceso de Erlang tiene un *Mailbox* que se puede accesar, en Io los actores tienen algo parecido, sin embargo es implícito.

- **Clojure**. Los modelos de concurrencia de Clojure y de Io son bastante similares. Los *Agents* de Clojure son similares a los actores de Io. Clojure también soporta *futures* y funcionan de una forma similar a los de Io. Clojure es un dialecto de Lisp y por lo tanto, es un lenguaje mucho más avanzado que Io. Por otra parte, en algunos casos Io puede ser más veloz que Clojure, ya que está escrito en C y el motor de concurrencia de Io es *SIMD*, mientras que la implementación principal de Clojure está hecha sobre la máquina virtual de Java que no está diseñada exactamente para ofrecer un buen soporte de concurrencia, como menciona Venkat Subramaniam en el capítulo de introducción de su libro.

6 Conclusiones

Io es un lenguaje que ofrece un modelo de concurrencia que a comparación de algunos otros lenguajes, es más sencillo de usar. Esto es una fuerte ventaja, sin embargo el problema que tiene Io es que no es un lenguaje muy eficiente**??**. Si bien es un lenguaje no tan veloz, su capacidad de concurrencia es una parte importante que lo hace poder competir con otros lenguajes más veloces. Es una lástima que no sea un lenguaje más usado, al punto de que no llega ni al top 50 del *TIOBE Index*. Esperamos que su popularidad aumente o que si no, al menos que lleguen nuevos lenguajes con ideas similares con respecto al modelo de concurrencia.

7 Agradecimientos

Queremos agradecer a Ariel Ortiz por su esfuerzo y dedicación en las materias que nos ha impartido. Ciertamente ha tenido una gran influencia en nuestras vidas como ingenieros en sistemas. También queremos agradecer a Steve Dekorte, por crear un lenguaje verdaderamente rebelde, que esperamos próximamente se vuelva famoso. A Bruce Tate por darle un lugar a Io en su libro y creer en él como un lenguaje que realmente puede cambiar la forma en la que uno piensa. De forma personal, Gerardo quiere agradacerle a su perra Byte, por portar el primer nombre computacional para perros.

Referencias

[1] Steve Dekorte *Io.*
http://www.iolanguage.com/ Accedido el 21 de octubre del 2012.

[2] Bruce Tate *Seven Programming Languages in Seven Weeks.*
http://pragprog.com/book/btlang/seven-languages-in-seven-weeks Accedido el 21 de octubre del 2012.

[3] Venkat Subramaniam *Programming Concurrency on the JVM: Mastering Synchronization, STM and actors* http://pragprog.com/book/vspcon/programming-concurrency-on-the-jvm Accedido el 16 de noviembre del 2012.

[4] Brian Foote *Class Warfare: Classes vs. Prototypes.*
http://www.laputan.org/reflection/warfare.html Accedido el 22 de octubre del 2012.

[5] Henry Lieberman *Using Prototypical Objects to Implement Shared Behavior in Object Oriented Systems* http://web.media.mit.edu/ lieber/Lieberary/OOP/Delegation/Delegation.html Accedido el 23 de octubre del 2012.

[6] Tiobe Software *TIOBE Programming Community Index for October 2012* http://www.tiobe.com/index.php/content/paperinfo/tpci/index.html Accedido el 23 de octubre del 2012.

Concurrencia en Modula-3

Salvador Aguilar (A00967057) Jorge Corona (A01164397)
Instituto Tecnológico y de Estudios Superiores de Monterrey
Campus Estado de México
Atizapán de Zaragoza, Estado de México, México.

31 de octubre, 2012.

Resumen

El objetivo de este artículo es analizar las ventajas y desventajas que nos presenta el lenguaje de programación Modula-3 para hacer programas concurrentes. Comenzaremos con un poco de historia sobre el lenguaje y por describir su sintaxis básica. Al ser un lenguaje un poco extenso sólo cubriremos lo necesario para poder analizar el uso de Threads que es el mecanismo por el cual se maneja la concurrencia. Al final trataremos de explicar como funciona la concurrencia en Modula-3 utilizando Threads

1 Un poco de historia

Modula-3 fue diseñado a finales de los años ochenta en Digital Equipment Corporation (DEC) y Olivetti Research Center (ORC) por Luca Cardelli, Jim Donahue, Mick Jordan, Bill Kalsow y Eric Muller. Modula-3 es un lenguaje miembro descendiente de la familia Pascal, es el sucesor inmediato de Modula-2+ y por su naturaleza por Modula-2. Mejorando muchas deficiencias de sus predecesores e incorporando nuevas funcionalidades, Modula-3 es un lenguaje de programación orientado a objetos, tiene manejo de excepciones, encapsulamiento, un recolector de basura automático y la característica principal de este artículo: manejo de Threads. Para esa época eran pocos los lenguajes de programación que implementaban el paradigma orientado a objetos y el recolector de basura automático, además Modula-3 es un leguaje bastante robusto por lo que sería interesante analizar por qué no es uno de los lenguajes más utilizados hoy en día, sin embargo, no es el objetivo de este artículo.

El objetivo principal del lenguaje era crear un lenguaje imperativo que implementara las características más importantes de los lenguajes modernos de ese tiempo de una manera sencilla y segura, es por esa razón que se omiten características como sobrecarga de operadores, herencia múltiple y otras características que son consideradas complicadas y "peligrosas".

	$Modula-3$	$Modula-2$
Excepciones	si	no
Genéricos	si	no
Threads	si	no
POO	si	no
Interfaces	si	si
Strong typing	si	no
Garbage collection	si	no

2 Generalidades del lenguaje

Modula-3 es un lenguaje imperativo, estructurado y modular. Un programa escrito en dicho lenguaje está compuesto por interfaces y módulos. Todas las interfaces y módulos utilizados por el programa se compilarán de manera individual y posteriormente se combinarán para formar el ejecutable.

Para empezar a introducir la sintaxis básica del lenguaje comenzaremos con el famoso "hola mundo" escrito en Modula-3.

```
MODULE Main;
IMPORT Wr, Stdio;
    (* Esto es un comentario en Modula-3 *)
    BEGIN
        Wr.PutText(Stdio.stdout, "Hello, World!\n");
    END Main
```

En el ejemplo anterior se utilizan dos interfaces: Wr y Stdio. Gracias a esas dos interfaces podemos llamar la instrucción PutText y usar la variable stdout. También podemos ver en el ejemplo anterior cómo se hacen comentarios en Modula-3. Sólo hay que comenzar el comentario con un paréntesis y un asterisco y terminar el comentario con un asterisco y un paréntesis. El módulo "Main" es por el que se comenzará a ejecutar el programa, es por eso que así fue como iniciamos nuestro Hola Mundo.

Para compilar nuestro Hola Mundo sólo es necesario teclear en nuestra terminal el siguiente comando:

```
m3 -o hello1 Hello1.m3
```

*Hay que considerar que nuestro programa se debe llamar "Hello1.m3"

2.1 Declaraciones, constantes y procedimientos

Vamos a comenzar esta sección con otro ejemplo que vamos a seguir utilizando a lo largo del texto. Dicho ejemplo nos ayudará a ejemplificar cómo se declaran constantes, variables y procedimientos. Además, posteriormente nos ayudará a ver la diferencia entre un programa escrito con Threads y uno escrito de manera secuencial.

```
MODULE CalcularPi EXPORTS Main;
IMPORT Wr, Stdio, Fmt;

CONST
    Rectas = 10000;

VAR
    medio: REAL
    alto: REAL
    ancho: REAL
    area: REAL
    suma: REAL

PROCEDURE Imprime(mensaje:TEXT) =
    BEGIN
        Wr.PutText(Stdio.stdout, mensaje);
```

```
        END Imprime;

    BEGIN
        suma := 0.0

        FOR i := 0 TO Rectas DO
             medio := (i + 0.5) * ancho;
             alto := 4.0 / (1.0 + medio * medio);
             suma := suma + alto;
        END

        area := ancho * suma;

        Imprime("El valor de Pi es " & Fmt.Real(area) & "\n";
    END CalcularPi.
```

Las Declaraciones se utilizan para proveer nombres (identificadores) a los objetos que se utilizan en el programa, en esta sección podemos incluir valores de literales (enteros, números de punto flotante, booleanos, caracteres, cadenas de caracteres mejor conocidas como strings). Como se muestra a continuación, las contantes deben comenzar con una llave o token CONST seguidas de una letra mayúscula. Con respecto a las variables, el token VAR va seguido de el nombre de nuestra variable. Recordemos que los nombres de variable deben comenzar con letras, seguidas del tipo de la variable, en este caso REAL, que es de tipo punto flotante:

```
CONST
    Rectas = 10000;

VAR
    medio: REAL
    alto: REAL
```

Los procedimientos tienen la misma intención que las funciones, encapsular una serie de declaraciones e instrucciones con una serie de parámetros que especifican información que se le pasará al procedimiento.

```
PROCEDURE Imprime(mensaje:TEXT) =
    BEGIN
        Wr.PutText(Stdio.stdout, mensaje);
    END Imprime;
```

Modula-3 no tiene facilidades integradas para entrada/salida por lo que es necesario utilizar procedimientos de varias interfaces con bibliotecas estándar para realizar operaciones de I/O. La interfaz Wr provee procedimientos para la salida a strings o caracteres. La interfaz Stdio define una salida estándar, stdout la cual esta destinada para la pantalla final del usuario. Para escribir números se utilizan procedimientos de la interfaz Fmt con el fin de convertir números a strings.

```
PROCEDURE Imprime(mensaje:TEXT) =
    BEGIN
        Wr.PutText(Stdio.stdout, mensaje);
    END Imprime;

(*LINEAS DE CODIGO *)

Imprime("El valor de Pi es " & Fmt.Real(area) & "\n";
```

Modula-3 tiene 4 instrucciones para utilizar ciclos WHILE, LOOP, REPEAT y FOR, la palabra EXIT se mantiene reservada para terminar los ciclos.

```
FOR i := 0 TO Rectas DO
    medio := (i + 0.5) * ancho;
    alto := 4.0 / (1.0 + medio * medio);
    suma := suma + alto;
END
```

2.2 Concurrencia en Modula-3 "Threads"

En los ejemplos anteriores, el programa se ejecutaba de manera secuencial, es decir, instrucción por instrucción. En nuestro próximo ejemplo con Threads vamos a tener varios puntos de ejecución en nuestro programa.

A partir de ahora vamos a comenzar a comparar el uso de Threads en Java y en Modula-3. La primera ventaja que encontramos para usar Threads en Modula-3 es que no es tan costoso como la creación de Threads en Java. Al igual que en Java, los Threads en Modula-3 comparten memoria lo que significa que pueden leer y modificar todas las variables, sin embargo, tienen su propio stack.

```
MODULE PiParalelo EXPORTS Main;
IMPORT Wr, Stdio, Fmt, Thread;

CONST
    Rectas = 10000;
    Ancho = 1.0 / Real(Rectas);

PROCEDURE Suma (inicio, fin : INTEGER) : Real =
    BEGIN
        VAR sum := 0.0;
        VAR mitad := 0.0;
        VAR alto := 0.0;

        FOR i := inicio TO fin DO
            mitad := (i + 0.5) * Ancho;
            alto := 4.0 / (1.0 + mitad * mitad);
            suma := suma + alto;
        END
    RETURN suma;
    END Suma;

PROCEDURE Imprime (mensaje : TEXT) =
    BEGIN
        Wr.PutText(Stdio.stdout, mensaje);
    END Imprime;

TYPE
    FHandle = Thread.T
    FClosure = Thread.Closure OBJECT
        inicio, fin : INTEGER
    OVERRIDES
        apply := RealizaSuma;
    END;
```

```
PROCEDURE IniciaSuma(inicio, fin : INTEGER) : Thread.T =
    VAR closure := NEW(FClosure, inicio := inicio, fin := fin);
    BEGIN
        RETURN Thread.Fork(closure);
    END IniciaSuma;

PROCEDURE RealizaSuma(closure:FClosure): REFANY =
    VAR result := NEW (REF REAL);

    BEGIN
        result ^:= Suma(closure.inicio, closure.fin);
        RETURN result;
    END RealizaSuma;

PROCEDURE EsperaSuma(handle: Thread.T) : REAL =
    BEGIN
        RETURN NARROW (Thread.Join(handle), REF REAL)^;
    END EsperaSuma;

BEGIN

    primerSuma := IniciaSuma(0, (Rectas / 2) - 1);
    segundaSuma := IniciaSuma(0, (Rectas / 2) - 1);

    resultado1 := EsperaSuma(primerSuma);
    resultado2 := EsperaSuma(segundaSuma);

    VAR total := resultado1 + resultado2;
    VAR area := total * Ancho;

    Imprime("El valor de Pi calculado con dos Threads es " & Fmt.Real(area) & "\n");

   END PiParalelo;
```

Como podemos ver en el ejemplo anterior, el proceso para utilizar Threads en Modula-3 es mucho más complejo que en Java. Sin embargo, es parecida la implementación del código que tiene que ejecutar cada Thread. En Java tenemos que sobre escribir el método "run". En Modula-3 sobre escribimos el método "apply" y le decimos que en vez de llamar "apply" debe de ejecutar el código que se encuentra en el procedure RealizaSuma. Una ventaja que le vemos a este tipo de implementación de Threads es que le puedes mandar parámetros a la función que se está sobre escribiendo a diferencia de Java que si queremos enviar parámetros la única manera que tenemos es agregar variables estáticas que todos los Threads pueden leer y modificar.

Para obtener el resultado final debemos esperar a que terminen de ejecutarse todos los Threads. Para eso utilizamos el procedure "EsperaSuma" que nos regresa el cómputo final del Thread que se le manda como parámetro. De esa manera nos aseguramos de que obtendremos el resultado esperado.

3 Conclusiones

Modula-3 es un lenguaje de programación que cuenta con la mayoría de las necesidades que hoy en día utilizamos. Es interesante ver que a pesar de ello, no es un lenguaje que se mencione mucho o que se utilice al mismo nivel que lenguajes que soportan lo que este lenguaje soporta, se menciona que puede ser un lenguaje más orientado a la enseñanza sin embargo uno de los problemas más grandes es que se le dejó de dar soporte al lenguaje conviertiéndose en un lenguaje olvidado. Es sin duda un lenguaje interesante para trabajar y para la época uno de los lenguajes que revolucionaron el concepto de la Programación Orientada a Objetos, pues como se mencionó se creó a finales de los ochenta y para 1991 no muy lejos se comenzaba a idear Java.

En cuanto al manejo de concurrencia, tiene implementados los mecanismos necesarios de sincronización para el manejo de Threads lo que nos permitiría escribir programas "Thread-safe", sin embargo, su implementación es un poco confusa a diferencia de otros lenguajes como Java. Probablemente en parte es porque ya estamos acostumbrados a desarrollar en Java que no es un lenguaje modular por lo que se nos hace más fácil instanciar objetos y escribir programas con uso de Threads.

Referencias

[1] Harbison, S. *Modula-3, 1st Edition.* Prentice Hall, 1992.

[2] Dr.Dobb's the world of software development *The Modula-3 Programming Language.* http://www.drdobbs.com/cpp/the-modula-3-programming-language/184409400 Accedido el 29 de octubre del 2012.

OpenCL, Programación concurrente y paralela

Arturo Ayala Tello (A01164742) Jorge de Jesús Tinoco Huesca (A01165318)
Instituto Tecnológico y de Estudios Superiores de Monterrey
Campus Estado de México
Atizapán de Zaragoza, Estado de México, México.

31 de octubre, 2012.

Resumen

Este documento es un artículo de investigación acerca del *framework* de programación OpenCL. En este artículo, se pretende describir a grandes rasgos sus características principales, conceptos y lenguaje de programación. Además, se incluye un poco de historia sobre la creación y concepción de la plataforma de programación y el modelo de concurrencia que ofrece.

1 Historia

OpenCL es un lenguaje diseñado para utilizar cualquier procesador que se encuentre, ya sea CPU, GPU u otros y fue diseñado principalmente por Apple Inc quien hizo partícipe del proyecto al grupo Khronos. Actualmente apoyan el proyecto diferentes OEMs (Original Equipment Manufacturer) como IBM, AMD, Intel, Nvidia, EA, Ericsson, entre muchos otros. Actualmente el proyecto le pertenece a Nvidia.

La primera versión que fue liberada al público fue el 8 de Diciembre de 2008, para la cual se trabajó 5 meses (de Junio de 2008 hasta Noviembre de 2008) y después el grupo Khronos revisó y aprobó el proyecto.

1.1 Khronos Group (El grupo Khronos)

Es un consorcio industrial sin fines de lucro que crea estándares abiertos para una gran variedad de plataformas y dispositivos de:

- Aceleración y cómputo paralelo.
- Gráficas.
- Medios dinámicos.
- Visión por computadora.
- Procesamiento de sensores.

Todos los miembros de Khronos son capaces de contribuir al desarrollo de las APIs (Application Programming Interfaces) de Khronos, las cuales son votadas antes de liberar alguna versión al público. Más de 100 compañías internacionales actualmente son miembros del grupo Khronos, teniendo como promotores a compañías como AMD, Apple, Nvidia, Intel, Sony Computer Entertainment, entre otros[1].

Algunos estándares abiertos por los cuales es conocido el grupo son:

- OpenGL.
- OpenAL.
- OpenGL ES.
- Collada.
- WebGL.
- Vision.

2 ¿Qué es OpenCL?

OpenCL significa *Open Computing Language* y su objetivo es hacer que máquinas heterogéneas de diferentes fabricantes puedan trabajar conjuntamente. OpenCL, básicamente, es un *framework* de programación para desarrollar aplicaciónes que aprovechen recursos computacionales heterogéneos y permite ejecutar código en plataformas mixtas (sin importar el fabricante o cuántos procesadores tiene) que pueden consistir en CPUs, GPUs y otros tipos de procesadores. Este *framework* incluye un lenguaje propio el cual mantiene similaridades con el lenguaje *C*. También hace uso de las GPUs para realizar tareas diferentes a gráficas computacionales (a esto se le llama *General Purpose GPU*).

OpenCL también se compone de un *API* que corre en la computadora anfitriona y hace posible el manejo y control de objetos y código de OpenCL, así como lidiar con los dispositivos de procesamiento viéndolos como unidades de procesamiento abstractas e independientes.

Sin embargo, se tiene que entender que OpenCL no proporciona los *SDKs*, éstos son proporcionados por la compañía correspondiente. Es decir, para el SDK de AMD tienes que ingresar al portal oficial de AMD y descargar el SDK de OpenCL para AMD, de igual forma para las tarjetas gráficas Nvidia.

2.1 ¿Por qué elegir OpenCL?

OpenCL es un lenguaje, sin embargo, no en toda la definición de un lenguaje. Mejor dicho, OpenCL es un conjunto de tipos, estructuras y funciones que pueden ser utilizadas en conjunto con el lenguaje *C* o *C++* y actualmente se han desarrollado versiones para *Java* y *Python*. OpenCL permite realizar tareas que con un lenguaje común aún no se puede, como lo es unir diferentes procesadores. Por ejemplo, con *C* o *C++* se puede programar para sistemas concurrentes con frameworks o bibliotecas como *TBB* u *OpenMP*, sin embargo, sólo se pueden utilizar CPUs. Con *CUDA* o con *Close To Metal* se pueden utilizar GPUs. La belleza de OpenCL radica en que puede hacer uso de ambos tipos de procesadores. Entonces, en general, las ventajas más significativas que brinda OpenCL sobre lenguajes como *C* o *C++* son:

- Portabilidad.
- Procesamiento estandarizado de vectores.
- Programación paralela.

2.1.1 Portabilidad

OpenCL adopta una filosofía similar a la de Java, pero con su propia versión: *"Write once, run on anything"*. Esto significa que sin importar la plataforma en la que se esté corriendo o si es CPU o GPU, no se tendrá que reescribir nada de código, ya que se utilizarían las mismas rutinas y funciones en todas las especificaciones de OpenCL. La portabilidad brinda también a OpenCL la capacidad de desarrollar aplicaciones con múltiples dispositivos como objetivo, donde estos dispositivos pueden tener diferentes arquitecturas o pueden estar fabricados por diferentes compañias. Lo único que se requiere en este tipo de aplicaciones es que los dispositivos acepten el *framework* de OpenCL.

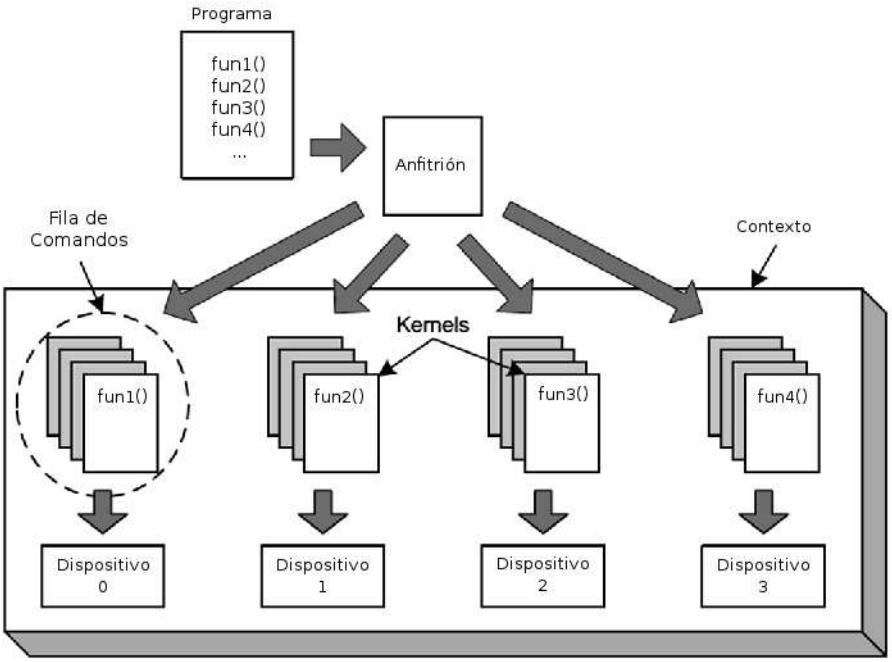


Figura 1: Distribución de kernels a través de los dispositivos

2.1.2 Procesamiento estandarizado de vectores

Las instrucciones para vectores generalmente son específicas para cada fabricante y éstas no tienen nada en común. Con OpenCL es posible programar rutinas para vectores y correrlas en cualquier procesador que las acepte, produciendo las respectivas llamadas específicas para cada tipo de dispositivo. Por ejemplo, el compilador de OpenCL para Nvidia producirá instrucciones PTX, mientras que en el de IBM, produce instrucciones AltiVec.

2.1.3 Programación paralela

La programación paralela se refiere a asignar tareas computacionales a diferentes elementos de procesamiento para ser realizados simultáneamente. Estas tareas en OpenCL son llamadas *kernels*. Un kernel es una función especial diseñada para ser ejecutada en uno o más dispositivos.

Para lograr esto, se tiene una aplicación principal (llamada *host*) que dispara los kernels a sus respectivos dispositivos. Es importante destacar que estos kernels pueden ser ejecutados tanto en el CPU donde se encuentra el host como en los demás procesadores heterogéneos.

El funcionamiento, a grandes rasgos, es el siguiente: La aplicación anfitriona maneja sus dispositivos a través de un contenedor llamado *contexto*. Existe otro contenedor de kernels (funciones) llamado *programa*. La aplicación dispara cada kernel hacía una estructura llamada *fila de comandos*. La lista de comandos es un mecanismo a través del cual la aplicación principal les indica a los dispositivos disponibles qué kernel va a ejecutar.

2.2 La especificación de OpenCL

El desarrollo de OpenCL, al ser tan dinámico y contar con la participación de un gran número de desarrolladores provenientes de diversas compañías, muestra su estado actual con mayor precisión dentro del sitio

oficial de OpenCL: www.khronos.org/opencl

Algo muy importante que podemos encontrar en este sitio web es la especificación para la versión más actual de OpenCL que exista en el momento de la visita. La especificación es por demás completa y muestra aspectos de gran relevancia para un programador interesado en adentrarse en el mundo de OpenCL.

La especificación define las funciones de OpenCL, sus estructuras de datos y también las características necesarias para poder desarrollar con las herramientas específicas de cada distribuidor de dispositivos. Así mismo, define los criterios necesarios para que estos dispositivos sean considerados como compatibles con el framework.

2.2.1 Extensiones

Además de las capacidades que brinda el uso de las bibliotecas estándar de OpenCL, la mezcla que se da entre *software* y *hardware* hace posible la creación de nueva funcionalidad. Estas nuevas características pueden ser disponibles para las aplicaciones de OpenCL a través de extensiones.

Las extensiones pueden ser específicas de un distribuidor o específicas de un dispositivo y el criterio que utiliza el grupo Khronos al momento de aprobarlas es el nivel de aceptación que ha recibido de la comunidad en general, lo cual muestra una vez más que el desarrollo conjunto es bien visto dentro del grupo. Dependiendo de la aceptación, cada extensión se nombra de diferente forma, mostrando a los programadores cuáles son aprobadas por el grupo en general y cuáles fueron liberadas por un distribuidor pero aún no han sido aprobadas.

3 Aspectos técnicos de OpenCL

Dado que OpenCL puede correr en diferentes plataformas, se tiene que tener un estándar de datos primitivos, ya que en un sistema un *int* puede ser de 32 bits y en otro sistema de 64 bits. Por lo tanto, OpenCL tiene sus datos primitivos, de los cuales mencionaremos algunos.[2]

Tipodedato	*Bits*	*Detalle*
`cl_char`	8	Entero con signo y complemento a dos
`cl_short`	16	Entero con signo y complemento a dos
`cl_int`	32	Entero con signo y complemento a dos
`cl_long`	64	Entero con signo y complemento a dos
`cl_half`	8	Punto flotante de precisión media
`cl_float`	32	Punto flotante de precisión simple
`cl_double`	64	Punto flotante de precisión doble

`cl_char`, `cl_short`, `cl_int`, `cl_long` también tienen la versión sin signo, y su nomenclatura es `cl_u[nombre]`.

3.1 Obteniendo información sobre las plataformas

Como se mencionó anteriormente, cada proveedor tiene los SDK propietarios (AMD tiene su SDK, Nvidia tiene su SDK, etc). Entonces, ¿cómo puedes crear una aplicación vendible si no sabes qué procesador utilizará tu cliente? Para este detalle OpenCL ofrece contar plataformas en lugar de saber en qué plataforma correrá el programa.

`cl_platform_id` es una estructura que detecta el número de *plataformas* que se tienen instaladas en la aplicación anfitriona. Lo que logra esta estructura es guardar la cantidad de SDKs que se tienen y saber exactamente cuál es.

```
int main() {
  cl_platform_id *platforms;
  cl_uint num_platforms;
  ...
  /* más codigo */
  ...
  err = clGetPlatformIDs(1, NULL, &num_platforms);
  if(err < 0) {
    perror("Couldn't find any platforms.");
    exit(1);
  }
  ...
  /* más codigo */
  ...
}
```

Así mismo, existen formas de obtener más información sobre la plataforma sobre la que el código de OpenCL va a correr. El método `cl_GetPlatformInfo` sirve para obtener este tipo de información. La firma del método es la siguiente:

`cl_int clGetPlatformInfo(cl_platform_id id, cl_platform_info name, size_t size, void *value, size_t *size_ret)`

El primer parámetro del método es de tipo `cl_platform_id`, que ya ha sido descrito previamente. El segundo parámetro es con el cual se elige el tipo de información que se desea obtener. Puede tener uno de los siguientes valores, predefinidos en OpenCL:

Nombre	*Propósito*
`CL_PLATFORM_NAME`	Regresa el nombre asociado con la plataforma
`CL_PLATFORM_VENDOR`	Identifica al distribuidor asociado con la plataforma
`CL_PLATFORM_VERSION`	Regresa el número de versión máximo soportado por la plataforma
`CL_PLATFORM_PROFILE`	Identifica el perfil de la plataforma, FULL_PROFILE o EMBEDDED_PROFILE
`CL_PLATFORM_EXTENSIONS`	Regresa una lista de extensiones soportadas por la plataforma

El uso se ve así:

```
char pform_vendor[40];
```

`clGetPlatformInfo(platforms[0], CL_PLATFORM_VENDOR, sizeof(pform_vendor), &pform_vendor, NULL);`

3.2 Obteniendo información sobre los dispositivos

El desarrollador puede necesitar, así como saber exactamente las características de la plataforma sobre la que correrá su aplicación, conocer los dispositivos que se encuentran disponibles para la misma y sus atributos específicos. OpenCL incluye funcionalidad para lograr estos objetivos.

De manera similar al método `clGetPlatformInfo`, existe también la función `clGetDeviceInfo` y funciona de la misma manera que su contraparte de información sobre la plataforma. Los parámetros que se le pueden enviar a la función, dependiendo de lo que se desee obtener son los siguientes:

Nombre	*Tipo*	*Propósito*
CL_DEVICE_NAME	char[ ]	Regresa el nombre del dispositivo
CL_DEVICE_VENDOR	char[ ]	Regresa el distribuidor del dispositivo
CL_DEVICE_EXTENSIONS	char[ ]	Regresa las extensiones del dispositivo soportadas por OpenCL
CL_DEVICE_GLOBAL_MEM_SIZE	cl_ulong	Regresa el tamaño de la memoria global del dispositivo
CL_DEVICE_ADDRESS_BITS	cl_uint	Regresa el tamaño del espacio de direcciones del dispositivo
CL_DEVICE_AVAILABLE	cl_bool	Indica si el dispositivo está disponible
CL_DEVICE_COMPILER_AVAILABLE	cl_bool	Regresa si la implementación tiene un compilador

Como podemos observar, el framework de OpenCL provee al programador de diversas herramientas para hacer que su aplicación realmente no dependa de la plataforma, ni de los dispositivos en los cuales va a correr. De esta manera, el desarrollador puede verdaderamente escribir su código una vez, tomando en cuenta las plataformas y dispositivos para los cuales desea que su aplicación corra y portarlo entre ellas de manera natural.

3.3 Partición de tareas

Una de las principales ventajas de utilizar OpenCL es la posibilidad de ejecutar aplicaciones que se lleven a cabo en un gran número de *threads* (hilos), llamados en este framework *work-items*. Para ilustrar el número de threads que se pueden usar en OpenCL, se puede imaginar una función que realice un ordenamiento de 2^{16} elementos enteros de 4 bytes. En este caso, el número total de work-items ideal sería $2^{16}/4$, es decir 2^{14}.

Los work-items, a su vez, son alojados en una estructura de OpenCL llamada *work-group*. Un work-group tiene un tamaño fijo de capacidad para cada plataforma. Sin embargo, si se crean más work-items de los que un work-group soporta, el framework se encarga de crear un nuevo work-group para darle cabida a los nuevos work-items.

También hay que considerar que cada work-item comparte memoria con los demás work-items del work-group. Por esto, OpenCL proporciona funciones para sincronizar a los work-items de un mismo work-group.

Es importante diferenciar entre los kernels y los work-items. Hay que recordar que un kernel en OpenCL es un conjunto de tareas que van a procesarse sobre cierta información o datos. Un work-item es una implementación del kernel en una porción específica de esos datos. Entonces, para un kernel pueden haber work-items múltiples.

4 Ejemplos prácticos

Para dar una breve pincelada de cómo se puede particionar una tarea en diferentes bloques, se puede ilustrar con tareas comunes de cualquier aplicación.

4.1 Paralelizando una instrucción *for*

Cuando se tiene una gran cantidad de datos estructurados, es común que se desee iterarlos para ejecutar alguna función sobre esos datos. Si se desea iterar sobre una estructura de datos multidimensional, es común utilizar ciclos anidados, los cuales hacen que la aplicación reduzca su velocidad de ejecución dramáticamente debido a su complejidad. Ejemplo:

```
for(i=0; i<x ; i++){
  for(j=0; j<y; j++){
    for(k=0; k<z; k++){
      procesar(arr[i][j][k]);
    }
  }
}
```

Esto se facilita y se hace eficiente con OpenCL [1] mediante la siguiente funcionalidad:

```
int i = get_global_id(0);
int j = get_global_id(1);
int k = get_global_id(2);

procesar(arr[i][j][k]);
```

El arreglo `arr[i][j][k]` es el *global ID* para un work-item. Este identifica cada work-item y le da acceso a la información que debe procesar.

4.2 Cómputo de π

Un ejemplo muy común es la implementación del cálculo de Pi. A continuación, se presenta una comparación de código escrito en C, contra código de OpenCL.

En este ejemplo se puede ver que, si bien sabemos que la aplicación escrita en OpenCL representará un aumento importante en la velocidad en la que se obtendrá el resultado, es verdad que el código de OpenCL no es nada sencillo de escribir y muchas veces tampoco de leer.

En este ejemplo se hace uso de la división de tareas. Se define el kernel para calcular pi y el work-item puede identificarse como el arreglo `out[ ]` que usa como *global ID* a la variable `i`.

Obviamente, el aumento en la velocidad del cómputo dependerá de los dispositivos y plataformas en las que sea corrido el programa, pero en una computadora convencional, con un procesador de dos núcleos, los tiempos de corrida fueron los siguientes:

Tiempo para calcular pi en la versión secuencial: 8.783 segundos

Tiempo para calcular pi en la versión OpenCL: 7.940 segundos

Código en C:

```
long num_steps = 100000000000;
double step = 1.0/num_steps;
double x, pi, sum = 0.0;

for(long i = 0; i<num_steps; i++){
  x = (i + 0.5) * step;
  sum += 4.0/(1.0 + x*x);
}
pi = sum * step;
```

Código en OpenCL:

```
#define _num_steps 100000000000
#define _divisor 40000
#define _step 1.0/_num_steps
#define _intrnCnt _num_steps / _divisor

__kernel void pi( __global float *out )
{
  int i = get_global_id(0);
  float partsum = 0.0;
  float x = 0.0;
  long from = i * _intrnCnt;
  long to = from + _intrnCnt;
  for(long j = from; j<to; j++)
  {
    x = ( j + 0.5 ) * _step;
    partsum += 4.0 / ( 1. + x * x);
  }
  out[i] = partsum;
}
```

4.3 MapReduce y otros acercamientos de paralelización

El framework teórico de MapReduce es un buen ejemplo de otros acercamientos que existen hacia la paralelización de tareas. Este acercamiento también es posible de programar usando OpenCL. Los work-groups son conceptos básicos para implementar MapReduce. La implementación divide la fase de reducción en dos subfases: reducción local y reducción global.[3]

El proceso de MapReduce en OpenCL se puede resumir en los siguientes pasos: (Figura 2)

- Cada work-item ejecuta el mapeo, pero en lugar de producir pares "llave-valor", también procesa una porción de la fase de reducción.
- El kernel sincroniza los work-items de manera que se previene más ejecución hasta que todos los work-items en un work-group hayan terminado.
- En cada work-group, el work-item con ID igual a cero reduce el output del work-group a un solo resultado.
- El kernel ejecuta una sincronización global que espera a que todos los work-groups terminen su ejecución.
- El work-item con ID igual a cero recibe el resultado de cada work-group y reduce estos datos para producir un resultado final.

MapReduce puede tomarse como ejemplo para mostrar que OpenCL es un framework que puede implementar diversos tipos de acercamientos hacia la paralelización, ya que su modelo de concurrencia es muy flexible.

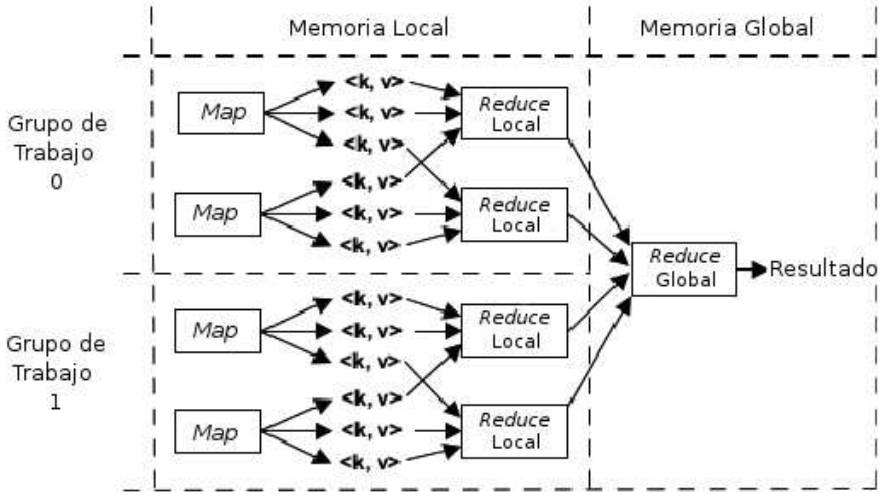


Figura 2: Funcionamiento de MapReduce en OpenCL.

5 Conclusiones

OpenCL es un tema complejo. Programar la aplicación más sencilla puede poner a prueba al desarrollador, ya que se necesita comprensión sobre programación en un anfitrión, programación para dispositivos y los mecanismos necesarios para transferir datos entre ambos. Sin embargo, si se logra dominar un framework tan poderoso como este, es evidente el jugo que se le puede sacar.

Además, un framework con un soporte tan grande, de parte de tantos distribuidores conocidos sencillamente brinda al programador la tranquilidad de que el framework está desarrollado de manera correcta, utilizando estándares de la industria de la tecnología de la información. Además de que es seguro de que los problemas serán solucionados velozmente, por un grupo altamente calificado de desarrolladores.

OpenCL es un framework extremadamente robusto, que tiene una funcionalidad muy extensa, imposible de describir completamente en un artículo como éste. Sin embargo, creemos que con lo descrito en él, alguien que desee conocer el funcionamiento general y los objetivos de OpenCL puede hacerlo al leer el presente artículo.

En general, programar en OpenCL "es como manejar un camión grande, de dieciséis llantas. Los principios de manejar son los mismos, pero al tener tanta carga en la caja, se tiene que lidiar y manejar pensando en muchas otras preocupaciones"[3].

Notas

[1] Claro, después de inicializar los objetos de OpenCL: contexto, dispositivos, anfitrión, plataformas, etcétera. Código que puede hacerse bastante extenso.

Referencias

[1] Khronos Group. *OpenCL - The open standard for parallel programming of heterogeneous systems.* http://www.khronos.org/opencl

[2] Aaftab Munshi. *OpenCL - Parallel computing on the CPU and GPU.* SIGGRAPH 2008

[3] Matthew Scarpino. *OpenCL in Action.* Manning Publications Co., 2012.

El lenguaje multiparadigma Oz

Gonzalo Landeros Valerdi (A00967875) Juan Manuel Roman Monterrosa (A00968306)
Instituto Tecnológico y de Estudios Superiores de Monterrey
Campus Estado de México
Atizapán de Zaragoza, Estado de México, México.

31 de octubre, 2012.

Resumen

El lenguaje de programación Oz, cuya implementación es el sistema llamado Mozart, tiene diversas características que lo hacen diferente a los demás lenguajes. En este documento se abarcará de manera general las características y funciones que ofrece este lenguaje multiparadigma, pero también se explorará de manera más específica su implementación de concurrencia la cual es a través del paso de mensajes.

1 Oz y Mozart

Oz es un lenguaje de programación el cual fue creado por Gert Smolka, junto con sus estudiantes en el año de 1991. En 1996 el desarrollo de Oz fue continuado gracias al grupo de investigación de Seif Haridi y Peter Van Roy en el Instituto Sueco de las Ciencias Computacionales.[1] Esa versión fue conocida como Oz 2.

A partir del año 1999 Oz, fue desarrollado por el grupo internacional conocido como el Consorcio Mozart. El grupo fue conformado por la Universidad del Sarre, SICS, y la Universidad Católica de Lovania. La implementación principal de Oz es el sistema Mozart, el cual fue liberado bajo una licencia de código abierto, por lo tanto Mozart y Oz son software libre, también conocido como *open source*. Actualmente Mozart implementa Oz 3, el cual está basado en un modelo concurrente y con restricciones.

2 Características

En términos generales Oz es un lenguaje de programación de alto nivel *multiparadigma*. Esto significa que soporta varios paradigmas de programación, a diferencia de otros lenguajes como C o Java que solamente utilizan uno. C utiliza el paradigma imperativo mientras que Java utiliza el paradigma orientado a objetos. Oz soporta varios paradigmas, entre ellos está el declarativo, orientado a objetos, imperativo, funcional, y el de programación por restricciones. Adicionalmente, Oz cuenta con las siguientes características:

- **Inferencia**. Esto se puede lograr ya que Oz soporta el paradigma declarativo y el paradigma por restricciones. El primero soporta árboles racionales, guardias profundos y el estilo Andorra no determinista. El segundo utiliza estrategias de búsqueda y distribución para definir las restricciones del árbol y cómo recorrerlo.
- **Distribución**. Es abierto, robusto y de red transparente. Muchos sitios pueden conectarse juntos de manera automática y se ejecutan en conjunto para conformar un solo bloque de instrucciones de Oz. Comparten variables, objetos, clases y procedimientos.

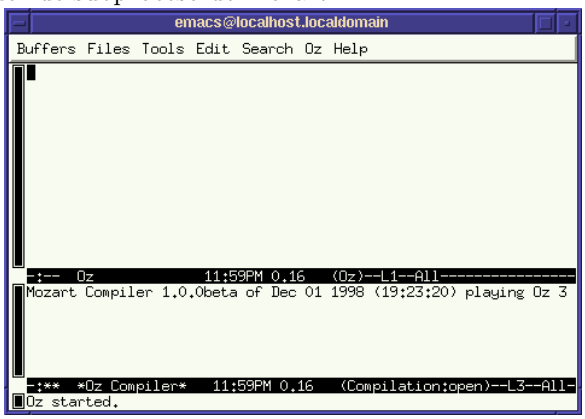
- **Concurrencia**. Se comunica utilizando el sistema de paso de mensajes asíncrono. Puede crear hilos, o *threads*. La gran diferencia que tiene con otros lenguajes es que cada hilo de Oz es un hilo de flujo de datos. Esto significa que solamente se ejecutará la declaración hasta que se resuelvan los conflictos de todas las variables dependientes de ella.
- **Interfaz gráfica o GUI**. Utiliza un enfoque tanto declarativo como imperativo para crear interfaces dinámicas fácilmente.

Mozart es un sistema implementado por Oz, por lo tanto es la combinación de diferentes áreas de este lenguaje. Es por esto que este sistema tiene soporte de aplicaciones distribuidas y de red. Es posible conectar varios cálculos de Oz ubicados en diferentes sitios para formar uno solo. Así mismo, Mozart puede transferir automáticamente los datos y el código entre sitios. Gracias a que es concurrente, tiene la capacidad de utilizar el paso de mensajes y compartir variables lógicas para la detección y manejo de errores que podrían perjudicar la red.

En este artículo se analizará con mayor profundidad el tema de concurrencia para poder apreciar detalladamente los beneficios que este sistema tiene. De esta forma se podrá hacer una comparación apropiada con otros lenguajes de programación. El alcance de este artículo no abarcará los demás tópicos, pero explicará con mayor detalle aquellos términos que estén relacionados con el tema principal.

3 Programando en Mozart

Mozart tiene un IDE conocido como OPI que significa *Oz Programming Interface*. El OPI utiliza el editor de Emacs para el programador. En esta interfaz se puede observar una ventana la cual está dividida en dos *buffers*. El buffer de arriba es un espacio donde el programador puede ejecutar piezas de código para observar su funcionamiento. El buffer de abajo es conocido como el compilador de Oz y despliega la interacción del programador con el compilador de subproceso de Mozart.[2]



Al comprender lo que hace cada buffer, uno puede comenzar a programar. Primero se programará el típico *hola mundo* escribiéndolo en el buffer de Oz de la siguiente forma:

```
{Show 'Hola Mundo'}
```

Para poder ejecutar este programa se debe posicionar el cursor sobre la línea escrita y seleccionar *Feed Line* del menú de Oz en la barra de menú. Al hacer esto podemos observar como se le alimentó al compilador esta línea y fue aceptada. El hecho de que haya sido aceptada significa que fue parseada y compilada. La salida del programa aparece en otro buffer conocido como el emulador de Oz. Este buffer contiene el transcript de ejecución y se puede ver en el menú *Show/Hide Emulator*.

Las llaves en el primer comando { ... } son usadas para procedimientos o llamadas a funciones. En este ejemplo podemos ver que *Show* es una función que contiene un argumento y en este caso el argumento es el átomo 'Hola Mundo'.

Anteriormente se había mencionado que Mozart tiene varias herramientas gráficas. La herramienta más conocida es el *browser*. Para poder invocar al browser uno deberá escribir lo siguiente:

```
{Browse 'Hola Mundo'}
```

Al ejecutar esta instrucción, una nueva ventana se abrirá con lo que se escribió. Esta ventana en particular puede ser muy eficiente debido a que se puede ver la forma en la que se van asignando las variables dentro de una función. Tomemos el siguiente código de ejemplo:

```
declare B A
{Browse foo(base:B altura:A area:thread B*A end)}
```

La salida de este programa nos da:

```
foo{
    base: B
    area: _
    altura: A)
```

Las variables base y altura han sido instanciadas, sin embargo aún no se les ha dado un valor, por lo tanto el valor del área está representado por un guión bajo. Si uno reemplazara el valor de **A** por un número, el área de cualquier forma no cambiará su valor. Si se les asignan valores tanto a **A** como a **B** el área se modificará a su valor correspondiente.

En estos códigos se puede observar el uso de variables. De acuerdo a Peter Von Roy, las variables son un atajo para declarar valores. No pueden ser asignados más de una vez. Pero sí se puede declarar una variable con el mismo nombre que tiene una variable previa. Sin embargo, esto impedirá poder acceder a dicha variable [5]. Afortunadamente, los cálculos hechos con la variable previa no se verán afectados por este cambio. Las variables tienen dos características importantes las cuales permiten estos comportamientos.

- Un **identificador**. Es la forma en la que el programador escribe la variable Debe empezar con una letra mayúscula y ésta puede ir seguida de una o más letras o números de cualquier tipo.
- Una **variable de almacenamiento**. Esta es la que el sistema utiliza para poder calcularla. Es parte de la memoria de Mozart la cual se le conoce como *store*.

Es importante mencionar que existe otra forma de crear variables y es utilizando los acentos graves de la siguiente forma:

```
`esta es una variable`
```

La palabra *declare* es utilizada para crear una nueva variable de almacenamiento y la une con su respectivo identificador. En resumen, si se utiliza el mismo identificador de una variable vieja para refereirse a una variable nueva, los cálculos de las variables viejas quedarán intactos, pero el valor de la variable será el de la nueva. Otra forma de poder hacer una declaración es a través de la palabra *local* en lugar de declare. Comparando los siguientes códigos:

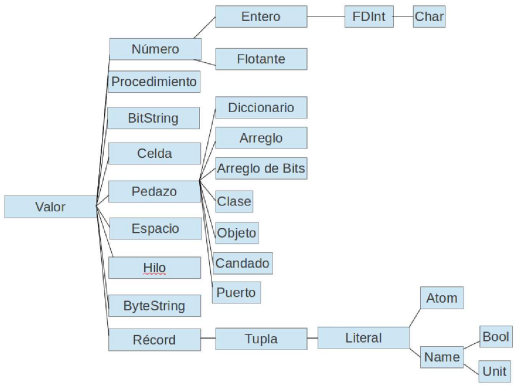
```
local X Y Z in S end
```

La palabra reservada local hace que las variables X, Y, y Z tengan un alcance o *scope* local.

```
declare X Y Z in S end
```

A diferencia de local, declare hace que las variables X, Y, y Z tengan un alcance global para todo el programa.

Oz es un lenguaje de tipado dinámico, por lo tanto cuando una variable es introducida tanto su tipo como su valor no son conocidos. La única forma de determinar su valor es unirla a través de un valor tipo Oz. A continuación se muestra un esquema de los diferentes tipos de valores que ofrece Oz:



La mayoría de los valores se pueden inferir para los programadores, sin embargo hay algunos que son diferentes como trozo, celda, espacio, FDInt, y nombre. A continuación se muestra una lista donde describen brevemente algunos de los tipos usados en Oz:

- **Números**. Pueden ser enteros o de punto flotante. Si son negativos se escriben con un guión, por ejemplo *10* se usa para expresar -10.
- **Átomos**. Son constantes simbólicas utilizadas para los cálculos. Muy parecidos a los átomos en lenguajes funcionales. Se escriben comenzando con letra minúscula, o entre comillas simples. Ejemplos son: *hola*, *atomo123*, y *´perro´*.
- **Booleanos**. Representados por *true* para verdadero y *false* para falso.
- **Registros**. Es una estructura de datos compuesta. Consiste de una etiqueta seguido de un par de características. Éstas pueden ser átomos, enteros, o booleanos. Un ejemplo es *person(age:X1 name:X2)*.
- **Tuplas**. Es un tipo de registro cuyas características son enteros consecutivos, comenzando desde el uno. Un ejemplo es este: *person(X1 X2)*. No hay necesidad de escribir las caracetrísticas a diferencia de los registros.
- **Cadenas de caracteres**. Se escriben con comillas dobles. Se ven representadas en una lista como numeros. Por ejemplo: *"ABC"* es lo mismo que *[65 66 67]*.
- **Trozo o Chunk**. Sirve para hacer tipos de dato abstracto, o *abstract data types*.
- **Celda o Cell**. Presenta lo que son los contenedores y modificadores de estado.
- **Espacio o Space**. Resuelven problemas avanzados de búsqueda.
- **FDInt**. Significa *Finite Domain Int* y es utilizado en la programación por restricciones.
- **Nombre o Name** introduce *tokens* que son únicos y anónimos.
- **Listas**. Consiste en el átomo *nil* o la tupla *(´|´H T)* donde T puede o no puede estar unida a una lista. Este ejemplo en particular es conocida como un *cons* o par de listas. El carácter *´|´* se escribe como un operador en infijo A continuación se muestran otros ejemplos:

```
Lista completa: [1 2 3] es lo mismo que 1|2|3|nil.
1|2|3|nil es lo mismo que 1|(2|(3|nil)).
```

- **Procedimientos**. Es un valor de tipo procedimiento. Tomando en cuenta el siguiente ejemplo:

  ```
  proc{P X1 X2 ... Xn} S end
  ```

  Aquí se crea el valor del procedimiento y la variable P es ligada a él. Un procedimiento en Oz tiene una identidad única dada por la unión que tiene ésta con una variable, por lo tanto cada procedimiento es diferente, a pesar de que parezcan ser iguales. Los procedimientos, threads, trozos y celdas tienen la característica particular de que la igualdad se ve representada en su nombre. Esto significa que si un procedimiento tiene el mismo nombre que otro, se igualan.[3]

3.1 Funciones

Las funciones en Oz pueden ser muy intuitivas para aquéllos que ya hayan programado en algún lenguaje funcional. Tomando el ejemplo de un factorial podremos apreciar la forma en la que se crean funciones.

A continuación se verán algunos ejemplos de lo que se puede hacer en Oz y Mozart.

```
declare
fun {Fact N}
   if N == 0 then 1 else N*{Fact N-1} end
end
```

Como bien ya se había dicho anteriormente, la palabra *declare* se utiliza para definir algo nuevo. La palabra *fun* es llamada para crear una nueva función **Fact**. El argumento de Fact es **N**. Debido a que es un argumento, N es una variable local y cada vez que se manda a llamar la función, una nueva variable N es declarada. Para este caso, N se irá reduciendo hasta llegar a su caso base que es cuando obtiene un valor de cero. En este mismo ejemplo se puede observar como Oz utiliza la recursión para calcular el factorial de un número.

Para probar si Fact realmente funciona se manda a llamar de la siguiente forma en el browser:

```
{Browse {Fact 10}}
```

Esto nos debería de dar de resultado **3628800**.

4 Concurrencia

La concurrencia en Oz se puede declarar de una manera más sencilla que otros lenguajes de programación. En los ejemplos anteriores se ha estado corriendo bajo un solo hilo. Lo primero que hay que hacer es importar el módulo de concurrencia llamado *Thread.this/1*. Los módulos en Oz son similares a los paquetes en *Java* o las bibliotecas en *C*. Si se tiene la referencia a un hilo el programador podrá realizar operaciones como terminar el hilo, o mandar a llamar una excepción dentro del mismo. Para crear un hilo nuevo se utiliza la siguiente instrucción:

```
thread S end
```

Al ejecutar esta instrucción, un hilo nuevo es dividido y corre de manera concurrente con el hilo actual. El hilo actual continúa con la siguiente declaración. A cada hilo se le concede una cantidad de tiempo del procesador. Este tiempo es distribuido de manera equitativa con los otros hilos. No obstante, el programador le puede asignar prioridades a los hilos para asignar más tiempo a un hilo que otro. Estas prioridades son bajas, medias, y altas. En Oz un hilo de alta prioridad no puede dejar en hambruna a un hilo de baja

prioridad. Esto se debe a que a todos los hilos les corresponde una porción de tiempo. La única diferencia es que el porcentaje de tiempo es menor.

El programa más frecuente que ha aparecido en varias fuentes, tanto en la página de Oz como en el libro de Van Roy, es el de creación exponencial de hilos utilizando la secuencia de Fibonacci.[2]

```
f X=<2 then 1
else thread {Fib X-1} end + {Fib X-2} end
end
```

Este código permite ver cuantos hilos de Oz soporta la computadora del programador.

4.1 Flujo de datos

Los hilos en Oz son de tipo *data-flow* o de flujo de datos. Esto significa que si existen dependencias de datos el hilo automáticamente se bloqueará. Esto se puede ver claramente en el código anterior. Al escribir esto en el Browser de Oz, uno podrá darse cuenta que las variables *X0, X1, X2, X3* no están ligadas a ningún valor. Es por esto que no continuará su ejecución y entrará en un estado bloqueado hasta que le manden un nuevo valor. Si se le asigna a *X0* un número, el hilo ejecutará la instrucción *Y0 = X0+1* y volverá a un estado bloqueado.

```
    declare X0 X1 X2 X3 in
    thread
        local Y0 Y1 Y2 Y3 in
        {Browse [Y0 Y1 Y2 Y3]}
        Y0 = X0+1
        Y1 = X1+Y0
        Y2 = X2+Y1
        Y3 = X3+Y2
        {Browse completed}
    end
end
{Browse [X0 X1 X2 X3]}
```

4.2 Condiciones de carrera y candados

Las condiciones de carrera también existen en Oz, debido a que el comportamiento de los hilos es no determinístico. Dado este comportamiento, un hilo puede llegar a ejecutarse antes que el otro, generando resultados inesperados. Analizando el siguiente ejemplo:

```
    declare
    C={NewCell 0}
    thread I in
        I=@C
        C:=I+1
    end
    thread J in
        J=@C
        C:=J+1
    end
```

A la variable C se le asigna un valor de 1 en cada uno de los hilos. Lo que se espera que contenga el valor de C es un total de 2, sin embargo debido a los tiempos de ejecución distintos de cada hilo, puede que no se detecte el incremento en el valor de C. El hilo I ve que C no ha sido modificado, así que asigna el valor de 1. Sin embargo el hilo J comenzó a ejecutrase antes de que I le asignara un valor a C por lo tanto considera que C tiene un valor de 0 y regresa 1.

La forma más adecuada de resolver este problema es a través de candados:

```
declare
C={NewCell 0}
L={NewLock}
thread
    lock L Then I in
    I=@C
    C:=I+1
    end
end
thread
    Lock L Then J in
    J=@C
    C:=J+1
    end
end
```

Agregar candados no es muy complicado y esto permite que no haya condiciones de carrera entre los hilos. Gracias a esto el resultado siempre será de 2.

Debido a que la concurrencia en Oz es un tema muy amplio se tocarán tres tipos de concurrencia de manera general: concurrencia declarativa, concurrencia de paso de mensajes y concurrencia compartida de estados.

4.3 Concurrencia declarativa

Los ejemplos de concurencia anteriores son de este tipo[4]. A continuación se muestran ejemplos de implementación que se pueden utilizar para este tipo de concurrencia:

- Utiliza sintaxis **declarativa**. Se pueden calendarizar procesos. Tiene esquemas de flujos productor-consumidor.
- Busca el **orden** de los cálculos. Se sabe los cálculos que tienen que hacerse, sin embargo debido a la dependencia de datos no se sabe el orden.
- Tiene **corrutinas** lo cual significa que son procesos coordinados entre sí manejados automáticamente por el sistema sin intervención del programador.
- Este modelo de concurrencia utiliza una técnica llamada *ejecución sobre demanda*, también conocida como *Lazy Execution*. Contar con este modelo de programación permite la existencia de:
  - *Disparadores sobre demanda*
  - *Lazy Functions*
  - *Manejo eficiente de memoria*
  - *Manipulción de archivos sobre demanda*

4.4 Concurrencia de paso de mensajes

Este tipo de concurrencia es similar a la del lenguaje de programación Erlang. Sus características son las siguientes:

- Manejo de **puertos**. Es un tipo de dato abstracto, o *Abstact Data Type* que permite la comunicación entre varios procesos de forma asíncrona, es decir, no espera respuesta de que el mensaje haya sido enviado. Utiliza la operación *send* para enviar un mensaje a otro hilo. El puerto necesita saber la dirección o lugar donde está dicho hilo.
- **Protocolos de mensaje**. El ejemplo más conocido es el *Remote Method Invocation*. Se invoca de forma distribuída un método que se encuentre adentro de un objeto. Esto puede ser de forma asíncrona o síncrona. Es similar al modelo cliente-servidor donde el servidor tiene definidos todos los métodos y al cliente únicamente los invoca.
- Uso de **Corrutinas**. Esto significa que son procesos coordinados entre sí manejados automáticamente por el sistema sin intervención del programador.
- *Data-driven model* o modelo manejado por datos. Esto da origen a lo que es conocido como la ejecución manejada por datos o *Lazy Concurrency*. Una característica de este modelo, es que cada hilo de ejecución no afecta la ejecución de los demás. Si en algún momento es necesario compartir un medio lo hacen de manera intercalada, sin afectarse.

4.5 Concurrencia compartida de estados

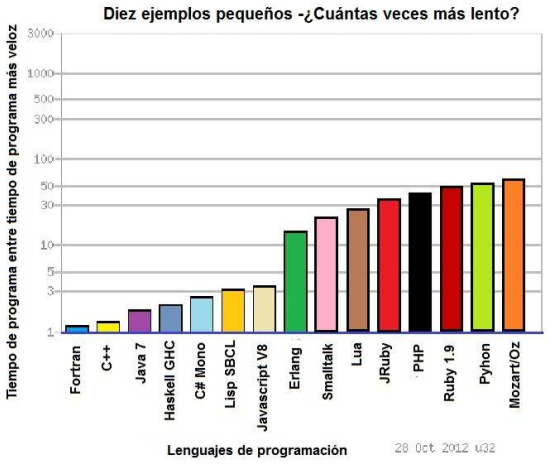
Este modelo es considerado como el más difícil de programar.

Es considerado como una extensión al modelo declarativo concurrente. Se agegan estados en forma de celdas y son un tipo de variable mutable. Este tipo de concurrencia también tiene similitudes con la del paso de mensajes debido a que puede ser implementado con puertos. No obstante, este modelo de concurrencia es más difícil de programar.

- Ejecuta múltiples hilos de manera independiente accediendo a **celdas** compartidas con operaciones atómicas. Una celda es un tipo de dato utilizado para definir el estado de un programa.
- Los hilos actualizan objetos pasivos compartidos a través de **acciones atómicas de granularidad gruesa**.
- Consiste en un conjunto de hilos accediendo a un conjunto de objetos. Esto se hace para limitar el número de intercalado de hilos.
- Tiene un **paso de mensajes** asíncrono.

5 Comparación con otros lenguajes

Habiendo explorado las funciones y características que Oz ofrece, se buscó comparar la velocidad de ejecución que tiene comparada con otros lenguajes. Se presenta la información obtenida a partir de la ejecución de diferentes algoritmos [5], condensados en una gráfica que permite visualizar los resultados de manera más sencilla. La gráfica representa el tiempo en que se ejecuta el programa, dividido entre el tiempo de ejecución del programa más rápido.



6 Conclusiones

Después de haber revisado y analizado las diversas características del lenguaje se concluye que Oz posee diversas características interesantes sobre el manejo de programas de forma concurrente, como implementación de esturcturas de datos complejas y esquemas de cómputo distribuído que posiblemente han sido de cierta influencia en otros lenguajes de programación o han tenido efecto en el desarrollo de varias bibliotecas que presentan muchas similitudes con diversas características del lenguaje como manejo de estructuras de datos concurrentes, siendo un ejemplo de ello Intel Threading Building Blocks.

7 Agradecimientos

Agradecemos al profesor Ariel Ortíz Ramírez en darnos la oportunidad de aprender sobre la importancia de la concurrencia en los lenguajes de la programación. Así mismo le agradecemos también por ayudarnos en practicar el uso de LaTeX.

Notas

[1] Swedish Institute of Computer Science o SICS.

[2] Debido a que tanto Mozart como Oz corren exclusivamente para computadoras con arquitectura de x86-32, la instalación del lenguaje de programación no fue posible. Se recurrirán a ejemplos de código obtenidos de manuales, presentaciones, y libros para facilitar el entendimiento del lector. Se ofrece una disculpa de antemano.

[3] Analizar de manera profunda estos tipos de variables no está dentro del alcance de este documento.

[4] La única excepción es el ejemplo de los candados ya que utiliza un modelo de concurrencia de estado compartido al utilizar variables de acceso (@), y asignación (:=)

[5] Los algoritmos usados son: pidigits, árboles binarios, k-nucleotide, spectral-norm, reverse-complement, n-body, fasta, mandelbrot y regex-dna

Referencias

[1] *Computer language benchmarks game* http://shootout.alioth.debian.org/ Accedido el 28 de octubre del 2012.

[2] Van Roy, P. Seif, H. (2003, 05) *LaTeX: Concepts, techniques, and models of computer programming*

[3] Collet, R.(2007, 12) *The limits of network transparency in a distributed programming language* http://www.info.ucl.ac.be/ pvr/raphthesis.pdf/ Accedido el 28 de octubre del 2012.

[4] Mejías, B.(n.d.) *Mozart-Oz Multi-paradigm Programming System* http://www.info.ucl.ac.be/ pvr/mozart-oz.pdf/ Accedido el 28 de octubre del 2012.

[5] Van Roy,P.(2006, 05) *How to say a lot with a few words* http://www.info.ucl.ac.be-/ pvr/GeneralOverview.pdf/ Accedido el 28 de octubre del 2012.

[6] Van Roy,P.(2002, 01) *Robust distributed programming in the Mozart platform: the importance of language design and distributed algorithms* http://www.info.ucl.ac.be/ pvr/lmo2002.pdf/ Accedido el 28 de octubre del 2012.

[7] http://www.mozart-oz.org/ Accedido el 28 de octubre del 2012.

Scala: Un lenguaje scalable

Edgar Mackey Vázquez Mejía (A01166320) Ademir Correa Loo (A01167255)
Instituto Tecnológico y de Estudios Superiores de Monterrey
Campus Estado de México
Atizapán de Zaragoza, Estado de México, México.

31 de octubre, 2012.

Resumen

Este artículo empieza explicando aspectos básicos de Scala: instalación, compilación de un programa, características especiales por las que es considerado un lenguaje multiparadigma, y cómo es su proceso de compilación. Más adelante enfocaremos nuestra atención a aspectos de la programación multinúcleo, mencionando qué herramientas utiliza para ello, así como también atacaremos el ejemplo del cálculo de Pi obteniendo el *speedup* y comparando el tiempo de ejecución en paralelo contra los de otros lenguajes como Java y Erlang. Al finalizar presentaremos nuestros agradecimientos y las conclusiones a las que llegamos.

1 Introducción

En este artículo se presenta a **Scala**, un lenguaje de programación multiparadigma, como una opción atractiva para el desarrollo de software paralelo. Está dirigido a personas con conocimientos de lenguajes orientado a objetos y que tengan noción sobre programación concurrente y/o paralela.

Scala fue diseñado por Martin Odersky y liberado a finales del 2003 e inicios del siguiente año. Los sistemas de tipo que utiliza son *static*, *strong*, *structural* e *inferred*. Este lenguaje de programación fue influenciado por Eiffel, Erlang, Haskell, Java, Lisp, Pizza, Standard ML, OCaml, Scheme y Smalltalk; y a la vez, ha influenciado a otros como Fantom, Ceylon y Kotlin. Corre sobre las plataformas JVM y CLR (Common Language Runtime).

2 Desarrollo

2.1 Hola mundo en Scala

Como un primer ejemplo, escribiremos el programa *Hola Mundo*.

```
object HolaMundo {
        def main(args: Array[String]) {
                println("Hola, mundo!")
        }
}
```

Este programa consiste de un método llamado `main` el cual toma los argumentos de línea de comando como un array de objetos String. El cuerpo de este método consiste en una llamada al método predefinido `println`.

El método `main` no retorna un valor porque se lo toma como un *procedure*, por ello no es necesario declarar un tipo de *return*.

Pero ¿qué hay con la declaración de `object` al inicio? Bueno, esa declaración introduce al objeto *singleton* que es una clase con una sola instancia. Por ello, dicha declaración declara (valga la redundancia) una clase llamada HolaMundo y una instancia de esa clase también llamada HolaMundo.

Para compilar el ejemplo utilizaremos el comando `scalac`, el cual funciona como la mayoría de los compiladores. Toma un archivo fuente como argumento, algunas opciones y produce uno o varios archivos objeto. Los archivos objeto que produce son archivos `class` de Java estándar.

Si guardamos el programa anterior en un archivo llamado `HolaMundo.scala`, podemos compilarlo ejecutando el siguiente comando:

```
$ scalac HolaMundo.scala
```

Esto generará algunos archivos `class` en el directorio donde nos encontramos. Uno de ellos se llamará `HolaMundo.class` y contiene una clase que puede ser directamente ejecutada utilizando el comando `scala`:

```
$ scala HolaMundo
```

2.2 Proceso de compilación

El compilador fue escrito por Martin Odersky. Este maduro compilador ha demostrado que es muy confiable a lo largo de varios años de uso en ambientes de producción. La implementación de este compilador produce código de bytes que ejecuta cada bit tan bien como lo haría el código de Java equivalente. Sin embargo, el compilador de Scala presenta un pequeño inconveniente, su velocidad de compilación. Odersky habla sobre esto:

> Hay dos aspectos con la relación a la (falta de) velocidad del compilador de Scala.
>
> 1. Mayor sobrecarga de inicio.
>
> Scala en sí consiste de muchas clases las cuales tienen que ser cargadas y compiladas en tiempo de ejecución.
>
> 2. Velocidad de compilación más lenta.
>
> Scalac maneja alrededor de 500 a 1000 líneas/seg. Javac maneja aproximadamente 10 veces eso. Hay muchas razones por esto.
>
> La inferencia de tipos es costosa, particularmente si involucra búsquedas implícitas.
>
> Scalac realiza comprobación de tipos dos veces; una según las reglas de Scala y otra más luego de la limpieza de acuerdo con las reglas de Java.
>
> Además de la comprobación de tipos hay cerca de 15 pasos de transformación para ir de Scala a Java, los cuales ocupan tiempo.
>
> Típicamente Scala genera muchas más clases por tamaño de archivo que Java, en particular si se usan bastante los modismos funcionales. La generación de Bytecode y la escritura de clases toman tiempo.

Por otro lado, un programa en Scala de 1000 líneas de código puede corresponder a uno en

Java de 2000 a 3000 líneas, entonces cuando se cuenta en término de líneas por segundo, una parte de la lentitud tiene que equilibrarse con mayor funcionalidad por línea.

2.3 Características especiales

Scala es orientado a objetos

Scala es un lenguaje puramente orientado a objetos en el sentido de que todo es un objeto, incluyendo números o funciones.

Sistema de tipos unificado

En Scala, todos los tipos de datos heredan de la clase mayor `Any` cuyos hijos intermedios son `AnyVal` (tipos de valor, como `Int` y `Boolean`) y `AnyRef` (tipos de referencia, como en Java). Esto significa que la distinción que hace Java entre los tipos de datos como `Int` e `Integer` no está presente en Scala.

Scala es funcional

A diferencia de C o Java, pero similar a Lisp, Scala no hace distinción entre sentencias y expresiones. En sí, todas las sentencias son expresiones que se evalúan para obtener un cierto valor. Las funciones que en C o en Java se declararían con un tipo de regreso `void` y las sentencias que no regresan ningún valor (como un `while`), en Scala son consideradas que regresan el tipo `Unit`, que es un tipo de *singleton*. Las funciones y los operadores que nunca regresan nada regresan `Nothing`, un tipo especial que no contiene objetos.

Scala es considerado un lenguaje funcional en el sentido que toda función es un valor. Asimismo, provee una sintaxis ligera para la definición de funciones anónimas, soporta funciones de orden superior, permite funciones anidadas, soporta la currificación, incorpora tipos de datos algebraicos, tuplas y objetos y variables inmutables.

Debido a la inferencia de tipos, los tipos de las variables, de los valores que regresan las funciones y otras expresiones más pueden ser omitidos ya que el compilador se encarga de deducirlos.

Scala es de tipado estático

Está equipado con un sistema de tipado expresivo que soporta clases genéricas, clases internas y tipos abstractos, tipos compuestos, tipado explícito de auto-referencias, vistas y polimorfismo.

Scala es extensible

Provee una combinación única de mecanismos de lenguaje que facilitan la adición de nuevas estructuras de control o la creación de lenguajes de dominio específico (DSLs).

2.4 Scala paralelo

Scala utiliza colecciones paralelas como una de sus maneras de implementar programación en multinúcleo. Estas colecciones son clases concretas que nos provee Scala las cuales se mencionan a continuación:

Array Paralelo

El array paralelo mejor conocido como `ParArray` es un array como lo conocemos con la diferencia que para acceder a los elementos del arreglo utiliza *splitters* el cual divide al arreglo y crea nuevos indices actualizados un poco parecido a lo que hacemos para calcular Pi. También utiliza *combiners* que como su nombre lo dice son utilizados para combinar el trabajo de los *splitters* por lo cual pueden tener un trabajo más pesado al no saber el tamaño exacto del arreglo.

Vector Paralelo

Éste, al igual que el array, utiliza a los *splitters* y *combiners*, solo que los vectores son representados como árboles por lo tanto los *splitters* dividen en subárboles. Los combiners concurrentemente mantienen un vector de elementos y son combinados al copiar dichos elementos de forma "retardada". Es por esta razón que los

métodos tranformadores son menos escalables que sus contrapartes en arrays paralelos.

Rango Paralelo

Un rango paralelo es una secuencia ordenada separada por intervalos, es muy similar al rango secuencial ya que no utilizan constructores ni *combiners*. Para aprovechar la estructura se puede mapear elementos lo cual nos produciría un vector paralelo. En el siguiente ejemplo se muestra cómo crear un rango de este tipo.

```
(1 to 10 par) map ((x) => x * x)
```

Donde se obtienen todos los cuadrados de los números menores a 10.

Tabla Hash Paralelo

Las tablas hash paralelas almacenan sus elementos en un array subyacente en una posición determinada por el código *hash* del elemento respectivo. Las versiones mutables de los *hash sets* paralelos (`mutable.ParHashSet`) y los *hash maps* paralelos (`mutable.ParHashMap`) están basados en tablas *hash*.

2.5 Calculando Pi

```
object PiParallel extends App {
        import scala.collection.GenSeq

        val seqNumRects = Range(0, 10000 * 10000).view
        val parNumRects = Range(0, 10000 * 10000).par.view

        def calculatePi(range: GenSeq[Int]): Double =
                range.map{i => 4.0 * (1 - (i % 2) * 2) / (2 * i + 1)}.sum;

        def calculateTime[A <: AnyRef](calcPi: => Double, msg: String) {
                println(msg)
                val t1 = System.nanoTime
                val pi = calcPi
                val t2 = System.nanoTime
                println("\tPi aproximado:
                \t\t%s\n\tTiempo calculado: \t%s mseg".format(pi, (t2 - t1) / 1000000))
        }

        println("Procesadores disponibles == "+collection.parallel.availableProcessors)
        calculateTime(calculatePi(seqNumRects), "Secuencial:")
        calculateTime(calculatePi(parNumRects), "Paralelo:")
}
```

En el ejemplo anterior se muestra el famoso cálculo de Pi que hemos estado revisando a lo largo del curso, en él se puede observar que al inicio declaramos dos variables: una la hemos llamado `seqNumRects` el cual representa nuestros número de rectángulos para calcular el área bajo la curva, y la otra `parNumRects`, donde la diferencia con la anterior es que ésta hace referencia a una colección paralela.

Debemos notar que para calcular el valor de Pi, tanto en secuencial como en paralelo, se usa la misma función `calculatePi()`. Lo único que hace que esta función se comporte de una u otra forma es debido al tipo de parámetro que recibe: para la ejecución en secuencial recibe a la variable `seqNumRects`, pero para ejecución en paralelo se recibe a `parNumRects`.

Luego de compilarlo varias veces registramos un promedio de velocidades, obteniendo un tiempo secuencial y paralelo como sigue:

```
Lenguaje: Scala
Procesadores: 2
Secuencial:
Pi aproximado:  3.141592643589326
Tiempo calculado:  9092 mseg

Paralelo:
Pi aproximado:  3.1415926435898958
Tiempo calculado:  5002 mseg
```

Con estos valores calculamos

$$SP = \frac{T1}{Tp} = \frac{9092}{5002} = 1.817672931$$

Ahora realizamos lo mismo pero con Erlang.

```
Lenguaje: Erlang
Procesadores: 2
Secuencial:
Pi aproximado:  3.141592643589326
Tiempo calculado:  12922.99 mseg

Paralelo:
Pi aproximado:  3.141592633589931
Tiempo calculado:  6947.277 mseg
```

Obteniendo el siguiente *speedup*:

$$SP = \frac{T1}{Tp} = \frac{12922.99}{6947.277} = 1.86015183$$

Y por último hacemos lo mismo para Java.

```
Lenguaje: Java
Procesadores: 2
Secuencial:
Pi aproximado:  3.141592643589326
Tiempo calculado:  15726.00 mseg

Paralelo:
Pi aproximado:  3.141592633589931
Tiempo calculado:  9076.00 mseg
```

Obteniendo el siguiente *speedup*:

$$SP = \frac{T1}{Tp} = \frac{15726.00}{9076.00} = 1.73270163$$

Con todos estos cálculos podemos decir que la velocidad en paralelo respecto a la secuencial en todos los casos siempre fue mayor y si no es que llegó a ser aproximadamente el doble de rápido. Si analizamos el *speedup*, se puede notar que Erlang tuvo mejor respuesta que los demás lenguajes.

3 Conclusiones

Para terminar este artículo con broche de oro no nos queda más que comentar que la experiencia de trabajar con Scala por lo menos en dos ejemplos fue bastante agradable. La elaboración de este artículo nos sirvió

mucho para aprender lo básico de un nuevo lenguaje de una manera didática y entretenida. Nos dio mucho gusto que usando este lenguaje para resolver el problema del cálculo de Pi, pudimos obtener un tiempo de ejecución en paralelo mucho menor en comparación a los otros lenguajes que hemos estado utilizando hasta el momento.

4 Agradecimientos

Queremos agradecer a nuestro profesor del curso Ariel Ortiz, quien nos motivó a tomar conciencia de la importancia de la programación concurrente.

Referencias

[1] École Polytechnique Fédérale de Lausanne. *The scala programming language.* http://www.scala-lang.org/ Accedido el 20 de octubre del 2012.

[2] Taft, D. (2012, Abril 16). *Application development: Scala programming language.* http://www.eweek.com/c/a/Application-Development/Scala-Programming-Language-10-Reasons-Developers-Need-to-Check-It-Out-524898/ Accedido el 26 de octubre del 2012.