Intro Ceylon

Gavin King - Red Hat

profiles.google.com/gavin.king

ceylon-lang.org

Somos fans de Java

Antes de empezar

Si critico Java (o cualquier otro lenguaje) es para identificar problemas y buscar soluciones

Leo muchas críticas contra Java, y con muchas yo personalmente no estoy de acuerdo—pero no significa que Java no tiene ningún problema!

Qué es?

Un lenguaje de programacion

- Ejecuta en *máquinas virtuales*
- Específicamente, la máquina virtual de Java, o VMs de JavaScript
- Definido por una *especificación*
- Con una sintaxis que se ve convencional pero en realidad es bastante flexible
- Con un sistema de tipos elegante y extremadamente poderoso
- Con modularidad integrada
- Con su propio SDK y tipos básicos (modulo de lenguaje)
- Y excelentes herramientas

Pa' qué?

Se usa:

- En Java SE, con su propio entorno de módulos
- En cualquier contenedor OSGi: Eclipse, Apache Felix, WildFly, GlassFish, ...
- En Vert.x
- En Node.js
- En un navegador, con Common JS Modules (require.js)
- En un contenedor de Java servlets, con el comando `ceylon war` (obra en progreso)

Qué tal interop?

Interoperable con código nativo

- Se puede crear in modulo multiplataforma que ejecuta en las dos maquinas virtuales, dependiendo solamente de otros modulos multiplataformas escritos en puro Ceylon
- O, se puede crear un modulo que solo funciona en una de las dos maquinas virtuales e interopera con código nativo de la plataforma (Java o JavaScript)
- Interoperacíon con JavaScript es a través de tipado dinámico, o con una interfaz escrita en Ceylon que provee de tipos estáticos al API de JavaScript

Qué tiene de especial?

Unas cosas únicas

- Diseñado para uso multiplataforma—el lenguaje básico abstrae completamente los detalles de la máquina virtual
- Tipos genéricos reificados, acompañados por un metamodelo de tipado estático que nos deja accesarlos en tiempo de ejecución
- Tipos uniones y intersecciones—la base de inferencia de tipos sin ambigüedad y tipado sensitivo al flujo
- Representación y abstracción de tipos de funciones y tipos de tuplas dentro del sistema de tipos—sin explosión de interfaces de método único o F1, F2, F3, ...
- Un sistema de tipos sencillo y unificado, con mucha azúcar sintáctica para reducir verbosidad sin hacer daño a la legibilidad

Idiom #1

Idiom: funciones con resultados múltiples

Por ejemplo, esta operación retorna un `File`, un `Url`, o nada:

```
//Java
Object parsePath(String path)
        throws SyntaxException { ... }
```

Manejamos los diversos resultados utilizando `instanceof`, casting, y `catch`:

```
try {
    Object result = parsePath(path);
    if (result instanceof File) {
        File file = (File) result;
        return lines(file);
    }
    if (result instanceof Url) {
        Url url = (Url) result;
        return new Request(url).execute().getContent().getLines();
    }
}
catch (SyntaxException se) { return emptyList(); }
```

Idiom #1

Idiom: funciones con resultados múltiples

La función con mas que un solo resultado puede ser definido utilizando un tipo union:

```
File|Path|SyntaxError parsePath(String path) => ... ;
```

Manejamos los diversos resultados con `switch`:

```
value result = parsePath(name);
switch (result)
case (is File) {
    return lines(result);
}
case (is Url) {
    return Request(result).execute().content.lines;
}
case (is SyntaxError) {
    return {};
}
```

Idiom #1

Idiom: funciones con resultados múltiples

Tenemos la opción de agregar casos con union:

```
value result = parsePath(name);
switch (result)
case (is File|Url) {
    ...
}
else {
    ...
}
```

O, en cambio, utilizando if en lugar de switch:

```
if (is File|Url result
        = parsePath(name)) {
    ...
}
```

Idiom #2

Idiom: funciones que retornan null

Ejemplo: obtener un elemento de un mapa.

(No es nada mas que un caso especial de resultados múltiples!)

```
Item? get(Key key) => ... ;
```

Aquí `Item?` literalmente significa `Null|Item`.

```
value map = HashMap { “CET”->cst, “GMT”->gmt, “PST”->pst };

Timezone tz = map[id]; //not well-typed!
value offset = map[id].rawOffset; //not well-typed!

Timezone? tz = map[id];
value offset = (map[id] else gmt).rawOffset;
```

Para un tipo union de esta forma tan común tenemos azúcar sintáctica especial.

Idiom #3

Idiom: colecciones heterogéneos

Qué tipo tiene una lista que contiene `Integers` y `Floats`?

```java
//Java
List<Number> list = Arrays.asList(1, 2, 1.0, 0.0);
```

El tipo del elemento es ambiguo, entonces hay que ser explícito.

Aun perdimos información:

```java
Number element = list.get(index);
//handle which the subtypes of Number?
//don't forget that an out of bounds
//index results in an exception
```

Idiom #3

Idiom: colecciones heterogéneos

Con union y intersección, inferencia de tipos ya no es ambiguo!

```
value list = ArrayList { 1, 2, 1.0, 0.0 };
```

El tipo inferido del elemento es `Integer|Float`, resultando en el tipo inferido `ArrayList<Integer|Float>`, que es un subtipo de todo tipo a cual podemos legalmente asignar ArrayList.

No hay ninguna perdida de precisión!

```
Integer|Float|Null element = list[index];
//now I know exactly which cases I have to handle
```

Idiom #4

Idiom: uniones y streams

Ejemplo: el método `follow()` de `Iterable` agrega un elemento al inicio del stream.

```
{Element|Other+} follow<Other>(Other element)
        => { element, *this };
```

La sintaxis `{T*}` and `{T+}` es azúcar para la interfaz `Iterable`

El tipo que sale es exactamente correcto:

```
{String*} words = { “hello”, “world” };
{String?+} strings = words.follow(null);
```

(Aunque estoy escribiendo los tipos explícitamente, los pude haber dejado ser inferidos.)

Idiom #5

Idiom: intersecciones y streams

Ejemplo: la función `coalesce()` elimina `null` del stream.

```
{Element&Object*} coalesce<Element>({Element*} elements)
        => { for (e in elements) if (exists e) e };
```

Otra vez, el tipo que sale es exactamente correcto:

```
{String?*} words = { “hello”, null, “world” };
{String*} strings = coalesce(words);
```

(Otra vez, pude haber dejado que los tipos fueran inferidos.)

Idiom #6

Idiom: vacío contra no-vacío

Problema: la función `max()` puede retornar `null`, pero solo en caso de que el stream pueda ser vacío. Empezamos con esto:

```
shared Value? max<Value>({Value*} values)
        given Value satisfies Comparable<Value> { ... }
```

Y si ya sabemos en tiempo de compilación que no puede ser vacío? Se necesita otra función distinta?

```
shared Value maxNonempty<Value>({Value+} values)
        given Value satisfies Comparable<Value> { ... }
```

Horrible! No nos deja abstraer.

Idiom #6

Idiom: vacío contra no-vacío

Solución: la interfaz `Iterable` tiene un parámetro de tipo extra:

```
shared Absent|Value max<Value,Absent>(Iterable<Value,Absent> values)
        given Value satisfies Comparable<Value>
        given Absent satisfies Null { ... }
```

El tipo que sale es exactamente correcto. (Y puede ser inferido.)

```
Null maxOfNone = max {};                          //known to be empty
String maxOfSome = max { “hello”, “world” };  //known to be nonempty

{String*} noneOrSome = ... ;
String? max = max(noneOrSome);              //might be empty or nonempty
```

Idiom #7

Idiom: valores de retorno multiples

Por ejemplo, una operación que retorna un `Protocol` y un `Path`:

```
//Java
class ProtocolAndPath { ... }

ProtocolAndPath parseUrl(String url) {
    return new ProtocolAndPath(protocol(url), path(url));
}
```

Java *nos obliga* a definir una clase.

Idiom #7

Idiom: valores de retorno multiples

Se puede definir una función que retorna una tupla:

```
[Protocol,Path] parseUrl(String url)
        => [protocol(url), path(url)];
```

Ahora el que invoca la función puede extraer los valores individuales:

```
value protocolAndPath = parseUrl(url);
Protocol name = protocolAndPath[0];
Path address = protocolAndPath[1];
```

Qué tal otros índices?

```
Null missing = protocolAndPath[3];
Protocol|Path|Null val = nameAndAddress[index];
```

Idiom #8

Idiom: desplegando tuplas

Ahora queremos pasar el resultado de `parseUrl()` a otra función:

```
Response get(Protocol name, Path address) => ... ;
```

Utilizamos *desplegar*, `*`, como en Groovy:

```
value response = get(*parseUrl(url));
```

O, si no, podemos hacerlo indirectamente, utilizando `unflatten()`

```
Response(String) get = compose(unflatten(get), parseUrl);
value response = get("http://ceylon-lang.org");
```

Existe una relación profunda entre los tipos de funciones y los tipos de tuplas.

Idiom #9

Idiom: abstraer sobre tipos de funciones

Problema: la función `compose()` compone funciones de diversos tipos.

```
X(A) compose<X,Y,A>(X(Y) x, Y(A) y)
        => (A a) => x(y(a));
```

Pero esto no es tan general como puede ser!

Para las funciones de un solo parámetro va bien:

```
Anything(Float) printSqrt = compose(print,sqrt);
```

Luego consideramos funciones de parámetros múltiples:

```
value printSum = compose(print,plus);
```

Idiom #9

Idiom: abstraer sobre tipos de funciones

Solución: abstraer sobre un tipo de tupla desconocido.

```
X(*Args) compose<X,Y,Args>(X(Y) x, Y(*Args) y)
        given Args satisfies Anything[]
        => flatten((Args args) => x(y(*args)));
```

Un poco feo, pero funciona!

```
Anything(Float,Float) printSum = compose(print,plus);
```

Aunque no les parezca, esto es muy útil, y lo usamos mucho, por ejemplo, para el metamodelo y para `ceylon.promise`