The Ceylon Type System

Gavin King
Red Hat
`in.relation.to/Bloggers/Gavin`

About this session

- This is the sequel to “Introducing the Ceylon Project”
- I’m not going to talk about why we’re doing a new language
- I’m not going to talk about the basic syntax of the language
- Instead, I’m going to discuss the Ceylon type system: inheritance, generics, and operators
- Of course, I’m not going to have time to cover *everything* that’s interesting

Principles behind the type system

- There should be no "special" types i.e. no primitive or compound types that can't be expressed within the type system itself - "everything is an object"
- There shouldn't be any special functionality for built-in types that can't apply equally to user-written classes, e.g., operators, numeric promotions
- Everything should still work "how you expect" from Java / C / etc
    - Numeric types should still behave how they behave in other languages
    - sequences should behave like arrays
- The overall complexity of the type system must not be much greater
    - This means sacrificing some things that are occasionally nice: method overloading, wildcard types
- Great language design means leaving things out!

Closure and block structure

This is an attribute ...

```
Person person {
    Name name {
        return Name(“Gavin”, “King”);
    }
    return Person(name);
}
```

Closure and block structure

... this is a method ...

```
Person person(String firstName, String lastName) {
    Name name {
        return Name(firstName, lastName);
    }
    return Person(name);
}
```

Closure and block structure

... and this is a class.

```
Person(String firstName, String lastName) {
    Name name {
        return Name(firstName, lastName);
    }
}
```

Notice that they are not very different!

Closure and block structure

- The language has a recursive structure
  - generally, constructs which are syntactically valid in the body of a class are also syntactically valid in the body of an attribute or method
  - a declaration contained inside a block always receives a closure of other members of the block
- We think of a class as a function which returns a closure of its members to the caller
  - of course, a big difference is that a class defines a type - class members can be `shared`
- We use the terms “function” and “method” almost interchangeably
  - within the block that contains a method declaration, the method appears to be a function (you can call it without specifying a receiver)
  - outside of the block, a `shared` method appears to be a member and must be qualified by the receiving object

Operator polymorphism

- We think that true operator overloading is harmful
  - the temptation to overload a common symbol like + to mean something that has nothing to do with addition is overwhelming for most library authors
  - once you are using several libraries, it's really hard to tell what any particular occurrence of + means (the problem is worse if you also have type inference)
  - people end up defining operators like `@:+>` resulting in intriguing, executable ASCII art, not plain, readable code
- On the other hand, it's really annoying that we can't define + for `Complex` numbers in Java, or define == to mean `equals()`
- The solution: operator polymorphism
  - An operator is just a shortcut for a method of a built-in type
  - > means `Comparable.largerThan()`
  - + means `Numeric.plus()`

Operator polymorphism

```
shared class Complex(Float re, Float im = 0.0)
        satisfies Numeric<Complex> {
    ...
    shared actual Complex plus(Complex that) {
        return Complex(this.re+that.re, this.im+that.im);
    }
    ...
}

Complex x = Complex(1.0);
Complex y = Complex(0.0, 1.0);
Complex z = x + y;    //means x.plus(y)
```

This solution is a "middle way" - less powerful, but simpler and safer than true operator overloading.

Inheritance model

- There are only three kinds of type: classes, interfaces, and type parameters
  - There are no special annotation or enum types
- There are four kinds of relationship between classes and interfaces
  - A class `extends` another class
  - A class or interface `satisfies` zero or more interfaces
  - A class or interface may have an enumerated list `of` its subtypes

Inheritance model

Like in Java, a class may extend another class, and implement multiple interfaces.

```
shared class Character(Natural utf16)
        extends Object()
        satisfies Ordinal & Comparable<Character> {
    ...
}
```

The syntax `X&Y` represents the intersection of two types. The syntax `X|Y` represents the union of two types.

Interfaces and mixin inheritance

An interface may declare both `formal` and concrete members.

```
shared interface Comparable<in T> {

    shared formal Comparison compare(T other);

    shared Boolean largerThan(T other) {
        return compare(other)==larger;
    }

    shared Boolean smallerThan(T other) {
        return compare(other)==smaller;
    }
    ...

}
```

An interface may not declare initialization logic, which is the cause of ordering and diamond inheritance problems.

Refinement (overriding)

The `actual` annotation specifies that a member *refines* a supertype member.

```
shared class Character(Natural utf16)
        extends Object()
        satisfies Ordinal & Comparable<Character> {

    Natural nat = utf16;

    shared actual Comparison compare(T that) {
        return this.nat<=>that.nat;
    }

    ...
}
```

The <=> operator is called “compare”. It’s just a shortcut for the method `compare()` of `Comparable`.

“Switching” by type

- Type narrowing is often frowned upon in object-oriented programming
  - especially frowned upon is the practice of writing a big list of cases to handle the various subtypes of a type (addition of a new subtype breaks the case list)
  - usually, polymorphism is the right way to do things - each subtype overrides an abstract method
  - however, there remain some cases where a list of cases is the right approach - for example, if the code that handles the various cases is in a different module to the code that defines the cases
- Unfortunately, Java exacerbates the problem
  - the combination of `instanceof` followed by a typecast is verbose and error prone (the compiler cannot validate typesafety)
  - the compiler does not inform us when addition of a new subtype breaks the list of cases

Type narrowing

Many classes and interfaces have multiple subtypes.

```
abstract class Node<T>(String name) { ... }

class Leaf<T>(String name, T value)
        extends Node<T>(name) { ... }

class Branch<T>(String name, Node<T> left, Node<T> right)
        extends Node<T>(name) { ... }
```

But Ceylon has no C-style typecasts.

Type narrowing

The `case (is ... )` and `if (is ... )` constructs perform a type check and type cast in one step, thus eliminating the possibility of `ClassCastException`s.

```
Node<String> node = ... ;
switch (node)
case (is Leaf<String>) {
    leaf(node.value);
}
case (is Branch<String>) {
    branch(node.left, node.right);
}
else {
    somethingElse(node);
}
```

node is a `Leaf<String>`

node is a `Branch<String>`

`All we know about node` is that it is a `Node<String>`

The compiler forces the `switch` statement to contain an `else` clause to handle other subtypes.

Enumerated subtypes

A class or interface may specify an explicitly enumerated list of subtypes.

```
abstract class Node<T>(String name)
        of Branch<T> | Leaf<T> { ... }

class Leaf<T>(String name, T value)
        extends Node<T>(name) { ... }

class Branch<T>(String name, Node<T> left, Node<T> right)
        extends Node<T>(name) { ... }
```

The functional programming community calls this an algebraic *datatype.*

Enumerated subtypes

```
Node<String> node = ... ;
switch (node)
case (is Leaf<String>) {
    leaf(node.value);
}
case (is Branch<String>) {
    branch(node.left, node.right);
}
```

The compiler validates that `switch` statements contain either an exhaustive list of possible subtypes. or an `else` clause.

Typesafe enumerations

A toplevel `object` declaration defines a type with a single instance.

```
shared object true extends Boolean() {}
shared object false extends Boolean() {}

abstract class Boolean()
        of true | false
        extends Case() { ... }
```

A class with an enumerated list of instances is similar to a Java `enum`.

Typesafe enumerations

```
switch (x>0)
case (true) { ... }
case (false) { ... }
```

The compiler validates that `switch` statements contain an exhaustive list of instances. (Or an `else` clause.)

If you add or remove an enumerated instance of a type, the compiler will force you to fix every `switch` statement of that type.

Parametric polymorphism (generics)

- Java's system of parametric polymorphism is very powerful, but also very complex
  - Raw types are a gaping hole in typesafety
  - Wildcard types are an extremely powerful solution to the problem of covariance/contravariance, but extremely difficult to understand, and syntactically heavyweight
  - Type erasure doesn't mix well with overloading
  - There are many problems where we need to know the runtime value of a type parameter
- The solution:
  - eliminate raw and wildcard types
  - eliminate overloading
  - reify type arguments
  - support type parameter variance annotations

Reified generics

Like in Java, a class or method may have type parameters, which may have constraints.

```
void print<E>(E[] sequence) {
    if (is String[] sequence) {
        for (String s in sequence) {
            writeLine(s);
        }
    }
    else {
        Formatter f = getFormatter(E);
        for (E e in sequence) {
            writeLine(f.format(e));
        }
    }
}
```

The expression `E` here is similar to `E.class` in Java.

It's possible to test the argument of a type parameter at runtime.

Variance

Is `WeakReference<String>` assignable to `WeakReference<Object>`?

```
interface WeakReference<T> {
    shared formal T? get();
    shared formal void set(T t);
}

WeakReference<String> hs = ... ;
WeakReference<Object> ho = hs;
Object? o = ho.get();
ho.set(1);
```

`get()` is OK ... a `String` is an `Object`

`set()` is not OK ... an `Object` is not a `String`

Variance

A type may be *covariant* or *contravariant* in its type parameter. (Respectively `in` or `out`.)

```
interface WeakReferenceGetter<out T> {
    shared formal T? get();
}

interface WeakReferenceSetter<in T> {
    shared formal void set(T t);
}
```

Variance

The compiler validates member signatures to check that the type really does respect the declared variance.

```
interface WeakReferenceGetter<out T> {
    shared formal T? get(T t);
}
```

Compile error: not covariant

```
interface WeakReferenceSetter<in T> {
    shared formal T set(T t);
}
```

Compile error: not contravariant

Variance

Variance affects assignability.

```
WeakReferenceGetter<String> ps = ... ;
WeakReferenceGetter<Object> po = ps;

WeakReferenceSetter<Object> co = .... ;
WeakReferenceSetter<String> cs = co;


WeakReferenceGetter<Object> po = ... ;
WeakReferenceGetter<String> ps = po;

WeakReferenceSetter<String> cs = .... ;
WeakReferenceSetter<Object> co = cs;
```

Compile error: not assignable

Compile error: not assignable

Trust me: this is way easier to understand than wildcard types in Java!

Collections and variance

- An interface like `List<T>` should be covariant in `T` since we almost always want a `List<String>` to be a `List<Object>`
- Therefore, we need to split operations which mutate the list to a separate interface `OpenList<T>`
- It turns out that this is the right thing to do anyway - a major problem with using Java collections in APIs is that it is never clear what an operation like `add()` will actually do:
  - mutate the object that provided the `List`?
  - mutate the client's copy without mutating the original object?
  - throw an unchecked exception at runtime since the object that provided the list called `Collections.immutableList()`?

Generic type constraints

- There are four kinds of generic type constraint:
  - upper bounds (the most common type)
  - lower bounds (the least common type)
  - initialization parameter specifications
  - enumerated type constraints
- The last two don't exist in Java

Generic type constraints

An upper bound specifies that the type argument must be a subtype of the given type.

```
shared class TreeSet<out T>(T... elements)
        satisfies Set<T>
        given T satisfies Comparable<T> {
    ...
}
```

Note that the syntax for declaring constraints on a type parameter looks just like the syntax for declaring a class or interface.

Generic type constraints

An initialization parameter specification specifies that the type argument must be a class with the given parameter types.

```
shared S join<S,E>(S... sequences)
        given S(E... es) satisfies Sequence<E> {
    return S(JoinedSequence(sequences));
}
```

Then the type may be instantiated within the declaration. This is possible because Ceylon has reified generics.

Generic type constraints

An enumerated type constraint specifies that the type argument must be one of the enumerated types.

```
void print<T>(T printable)
        given T of String | Named {
    String string;
    switch (printable)
    case (is String) {
        string = printable;
    }
    case (is Named) {
        string = printable.name;
    }
    writeLine(string);
}
```

`T` is essentially a union type.

This is similar to an overloaded method in Java.

What next?

- We need help implementing the compiler and designing the SDK.
- This is a great time to make an impact!

Questions?