MATERIALS & SHAPES

Ben Greenman
December 16, 2014
OUTLINE

• Getting F-Bounded Polymorphism Into Shape
  • with Fabian Muehlboeck and Ross Tate, PLDI 2014
  • and the Ceylon team

• plus some more recent developments
MY GOALS

1. Explain the big discovery of the paper

2. Share the conclusions we drew

3. Convince you that we've acted sensibly
THE PROBLEM

• Type-safe equality in object-oriented languages

  • `Cat() == Animal()` ✓ Cast to common super
  • `42 == "forty-two"` ✗ Type error
  • `λx.42 == λx.42` ✗ Type error, undecidable*

Cast to common super

Type error

Type error, undecidable*
THE PROBLEM

- Type safe equality
  - List<T>
  - HashMap<T>
  - and so on ...
The state of the art? **Object.equals()**

```java
Class Object

public class Object

Class Object is the root of the class hierarchy. Every class has Object as a superclass. All objects, including array

Method Summary

<table>
<thead>
<tr>
<th>Modifier and Type</th>
<th>Method and Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>protected Object</td>
<td>clone()</td>
</tr>
<tr>
<td></td>
<td>Creates and returns a copy of this object.</td>
</tr>
<tr>
<td>boolean</td>
<td>equals(Object obj)</td>
</tr>
<tr>
<td></td>
<td>Indicates whether some other object is &quot;equal to&quot; this one.</td>
</tr>
<tr>
<td>protected void</td>
<td>finalize()</td>
</tr>
<tr>
<td></td>
<td>Called by the garbage collector on an object when garbage co</td>
</tr>
<tr>
<td>Class&lt;?&gt;</td>
<td>getClass()</td>
</tr>
<tr>
<td></td>
<td>Returns the runtime class of this Object.</td>
</tr>
</tbody>
</table>
```
The state of the art? `Object.equals()`
WHAT'S WRONG?

• Does not scale.
  • Should there be an `Object.compareTo()`?

• Masks errors that the static type-checker could find.

• The concept of "equality" is not defined for all objects.

• Requires dynamic dispatch
// Typical implementation

class Foobar extends Object {
    boolean equals(Object obj) {
        if (obj instanceof Foobar) {
            Foobar that = (Foobar) obj;
            /* Actually compare `this`
            * and `that` */
        }
        return false;
    }
}

× Wrong arg. type
× Run-time cast
× Dynamic check
× Lots of boilerplate
// It just gets worse

- `instanceof` checks show up everywhere
- Repetitive, many opportunities for bugs

```java
class BinaryTree<T> {

    boolean contains(T elem) {
        if (elem instanceof Comparable) {
            /* Implement me! */
        }
        return false;
    }
}

void remove(T elem) {
    if (elem instanceof Comparable) {
        /* Implement me! */
    }
}
```
We can do better!

- Ideally, declare an `interface`
  
  ```java
  Equatable<T> { boolean equalTo(T that); }
  ```

- Replace `instanceof` and casts with F-Bounded polymorphism
  
  ```java
  BinaryTree<T extends Equatable<T>> { ... }
  ```
An example: List

- Two lists are equal if their elements are pointwise equal.

```
List<T> extends Equatable<List<Equatable<T>>>
```

```
2
```

```
3
```

```
2.00 == 2.00
```

```
3.00 == 3.02
```

```
List<Integer>
```

```
List<Double>
```
An example: List

- Two lists are equal if their elements are pointwise equal.

\[
\text{List}\langle T \rangle \text{ extends Equatable}\langle \text{List}\langle \text{Equatable}\langle T \rangle \rangle \rangle
\]

List\langle Integer\rangle  List\langle Double\rangle
VARIANCE

• Read-only types are covariant (\textit{out, +, extends, ...})
  
  • A \texttt{List<Integer>} can safely be treated as a \texttt{List<Double>}

• Write-only types are contravariant (\textit{in, -, super, ...})
  
  • A \texttt{Consumer<Animal>} can be treated as a \texttt{Consumer<Cat>}

• Read-Write types are invariant
  
  • An \texttt{Array<String>} should contain exactly \textit{Strings}
Exception in thread "main"
java.lang.ArrayStoreException: Adult
An example: List

- Two lists are equal if their elements are pointwise equal.

\[
\text{List}\langle T \rangle \text{ extends Equatable}\langle \text{List}\langle \text{Equatable}\langle T \rangle \rangle \rangle
\]

![Diagram showing equality of lists with elements 2 and 3, and 2.00 and 3.00, and non-equality of 2.00 and 3.02.]
An example: List

- Two lists are equal if their elements are pointwise equal.

\[
\text{List}<T> \text{ extends } \text{Equatable}<\text{List}<\text{Equatable}<T>>> \\
\]

- List is covariant (we get elements out of it)
- Equatable is contravariant (we supply arguments)
An example: List

- Two lists are equal if their elements are pointwise equal.

```java
List<T> extends Equatable<List<Equatable<T>>>
```

This actually works!
THE BIG DISCOVERY

• The Ceylon team wanted to avoid `Object.equals()`

• Ross suggested the above solution

• Ceylon's response: **NO.**
THE BIG DISCOVERY

• "A List<Equatable<T>> is nonsense!"

• Lists contain data, but Equatable is an abstract concept.

| List<Integer>>                                      | <::| List<Equatable<Integer>>||>
|-----------------------------------------------------|----|-----------------------------|

1 2

Easy to imagine

Eq<1> Eq<2>

Not so easy to understand
THE BIG DISCOVERY

• "A List<Equatable<T>> is nonsense!"

• Lists contain data, but Equatable is an abstract concept.

Equatable is a constraint on Integers

Integers are a valid instantiation for List<T>

You never want a "list of constraints"
EXPERIMENT

- Ceylon is only one project. We weren't convinced.
- Surveyed 60 Open-Source Java projects
  - ~13.5 million lines of code (avg. 242,113 med. 60,062)
  - ~100,000 classes (avg. 1,962 med. 487)
  - ~10,000 interfaces (avg. 202 med. 41)
EXPERIMENT

You never want a "list of constraints"

- We can't tell what programmers were thinking
- Or they challenges they faced in development
- But, we can formalize Ceylon's opinion in the Java compiler without breaking backwards-compatibility
Types like `Equatable<Integer>` were never used as:

- Type Parameters
- Function arguments or return types
- Local variables or fields
What is a "type like" `Equatable<Integer>`?

- **Object**
  - **Animal**
    - **Billfish**
      - **Marlin**

```java
> class Object {
>     class Animal {
>         class Billfish extends Animal {
>             class Marlin extends Billfish {
> ```
What is a "type like" Equatable<Integer>?

> inter Equatable<T> {}
> class Swordfish extends Billfish, Equatable<Swordfish>
What is a "type like" `Equatable<Integer>`?
EXPERIMENT

(more precisely)

- Parameterized types used to complete cycles in the inheritance hierarchy were never used as:
  - Type Parameters
  - Function arguments or return types
  - Local variables or fields
RECAP

1. The problem: type-safe equality
2. Proposed solution: **Equatable** and F-Bounded Polymorphism
3. Strong Reject from industry
4. **Equatable** is a constraint, and causes cyclic inheritance

Next Up: the research perspective
The problem with `Equatable<List<...>>`
```java
> class List<T> extends Equatable
> inter Equatable<T> {}
> class Tree extends List<Tree>
```
-? List<Tree> <: Equatable<Tree>

```
> inter Equatable<T> {}
> class List
    extends Equatable
    <List <Equatable <T>>
> class Tree
    extends List<Tree>
```
-? Eq<List<Eq<Tree>>> <: Equatable<Tree>

```
> class List<T> {}
> class Tree extends List<Tree>
> inter Equatable<T> {}
```
-? Tree <: List<Equatable<Tree>>

```java
> class List<T> extends Equatable
> class Tree extends List<Tree>
> inter Equatable<T> {}
```
-? List<Tree> <: List<Equatable<Tree>>

```java
> class List
  extends Equatable
  <List <Equatable <T>>

> class Tree
  extends List<Tree>
```

Equatable

List

Tree

List Equatable

Equatable List

Object

Equatable<T>

> inter Equatable<T> {}

> class List
  extends Equatable
  <List <Equatable <T>>
>

> class Tree
  extends List<Tree>
-? Tree <: Equatable<Tree>  

Cycle!

```java
> class Tree
  extends List<Tree>

> class List
  extends Equatable

> inter Equatable<T> {}

> Equatable List

> Equatable<T>

List Equatable

Equatable List

Object

Tree

List<T>

Equatable
```
PRIOR WORK

• On the Decidability of Nominal Subtyping with Inheritance
  • Andrew Kennedy & Benjamin Pierce, FOOL 2007

• The general problem is undecidable

• Can recover decidability by removing either:
  1. Contravariance
  2. Expansive Inheritance
  3. Multiple Instantiation Inheritance*
1. Remove Contravariance

For all types $c<*>$, $d<*>$, and all values $x$, $y$:

$$c<x> \text{ is a subtype of } d<y>$$

if

$$x \text{ is a subtype of } y$$
PRIOR WORK

2. Remove Expansive Inheritance

Suppose \( C\langle X \rangle \) inherits \( D\langle Y \rangle \),

Either \( X = Y \)

or

\( X \) does not appear in \( Y \)

(\( Y \) is no "larger" than \( X \))
3. Remove Multiple Instantiation Inheritance*

For all types $c, d^{<*>}$, and all values $x, y$:

$c$ cannot inherit both $d^{<x>}$ and $d^{<y>}$

* All expansive-recursive type parameters must be invariant and linear
PRIOR WORK

• Taming Wildcards in Java's Type System
  
  • Ross Tate, Alan Leung, Sorin Lerner, PLDI 2011

No nested contravariance in:

  inheritance clauses
  or
  type parameters
List<T> extends Equatable

Contravariance

Nested Contravariance

Expansive Inheritance

Bad design
List<T> extends Equatable

Programmers separate *data* from "constraints on data". This separation leads to decidable subtyping.

❌ Nested Contravariance  ⊆  ❌ Bad design
Programmers separate data from "constraints on data". This separation leads to decidable subtyping.
Programmers separate data from "constraints on data". This separation leads to decidable subtyping.

**Materials**

- Object
- List<T>
- Swordfish

Cycle-free inheritance

**Shapes**

- Equatable<T>
- Cloneable<T>
- Addable<T>

Never used as type parameters
SUMMARY

• While studying type-safe equality, we found a strange pattern

  • **Equatable, Comparable, Hashable** are different!

• Following this *pattern* intuitively gives decidable subtyping

• These **Shapes** describe the structure and constraints of data

• In contrast, **Materials** are the data used and exchanged
MATERIALS & SHAPES
SUB-GOALS

i.e. "where can we go from here?"

1. Decidable subtyping
2. Type equality, decidable joins
3. Conditional inheritance
4. Shape shifters
WELL-FOUNDED INHERITANCE

• Undecidability results were caused by cyclic inheritance
  • Impossible to predict how type parameters would expand
• Without shapes, inheritance is well-founded
  • No more cycles!
  • An object's inheritance graph is known at compile-time
• Many applications
DECIDABLE SUBTYPING

• Strategy: define a measure on judgments $x <: y$

• Key idea: inheritance never introduces new shapes

• Two components:
  
  • The number of shapes appearing in each type
  
  • The maximum number of proof steps until the next shape
TYPE EQUALITY

• Suppose the type system has intersection types, $X \& Y$

• Is $\textbf{List}\langle X \& Y \rangle$ equal to $\textbf{List}\langle Y \& X \rangle$? (It should be!)
  
  • Not true in Java
  
  • Not true using Kennedy & Pierce's technique
  
  • Not true using Tate et al.'s technique
TYPE EQUALITY

- Our subtyping algorithm only depends on recursion
  
  - Never uses syntactic equivalences

- We get equality for free: \((A = B)\) iff \((A <: B)\ and \(B <: A\)\)
JOINS

• $A \sqcup B$ is the least common supertype of $A$ and $B$

• Useful for type-checking conditional statements.

  • `if (C) then A else B` has type $A \sqcup B$

• In many languages, arbitrary joins do not exist
interface Addable<out T> {}  
class Double implements Addable<Double> {}  
class Integer implements Addable<Integer> {}

Addable<??>  
  Addable<out Addable<??>>  
    Addable<out Addable<out Addable<??>>>  
      Integer  
      Double
JOINS

• Our system: the join of two materials always exists
  • Because material inheritance is decidable
• Note: Addable<*> was never the desired result
  • The result of any computation must be a material
CONDITIONAL INHERITANCE

• Unanswered question: type-safe equality for `List<T>`

• First solution, again: `List<T> extends Eq<List<Eq<T>>>`

  • Bad style

  • Nested contravariance & expansive inheritance

  • List elements **forced** to extend `Eq` -- cannot make a `List<Object>`
CONDITIONAL INHERITANCE

• Ideally, \texttt{List<T>} is \textcolor{red}{Equatable} if and only if its elements are

\[
\text{List<out T> satisfies Equatable given T satisfies Equatable}
\]

• "\textcolor{red}{satisfies}" indicates that shapes are constraints, orthogonal to material classes and interfaces

• "\textcolor{red}{given}" denotes a condition that holds for certain instances
CONDITIONAL INHERITANCE

• Surprisingly challenging! Consider:

```java
> interface List<out T> satisfies Cloneable
given T satisfies Cloneable

> class Array<inv T> extends List<T>
satisfies Cloneable
given T satisfies Cloneable

> class B satisfies Cloneable

> class A extends B
```

• What is the result of invoking `Array<A>.clone()`?
SHAPE SHIFTERS

• Code reuse is fundamental to object-oriented programming

• Shapes express constraints at the class / interface level

• **Shape Shifters** are a proposal for type variable-level reasoning

Set<String with CaseInsensitive>

Set<Function<Int, Int> with RefEqual>
The End