Integrating Coercion with Subtyping and Multiple Dispatch

Victor Luchangco

Programming Languages Research Group
Sun Microsystems Laboratories

ACM Symposium on Applied Computing
18 March 2008

(joint work with Joe Hallett, Sukyoung Ryu, Guy Steele)

```
4*pi*r**3/3
```

```
4*pi*r**3/3
```

```
REAL(4)*pi*REAL(r**3)/REAL(3)
```

```
class Node {
   Object item;
   ...
   Node(Object x) { item = x; }
   ...
}
Node n = new Node(3);
```

```
class Node {
   Object item;
   ...
   Node(Object x) { item = x; }
   ...
}
Node n = new Node(3);


Node n = new Node(Integer(3));
```

Coercion reduces clutter

```
Sub printStr(s As String, n As Integer)
  For i As Integer = 1 to n
    Console.Write(s)
  Next i
End Sub
```

Adapted from Peterson

- What does `printStr(7, "4")` do?

- What does `printStr(7, "4")` do?

Writes `"7"` 4 times to `Console`.

```
Sub printStr(s As String, n As Integer)
  For i As Integer = 1 to n
    Console.Write(s)
  Next i
End Sub
```

- What does `printStr(7, "4")` do?

Writes `"7"` 4 times to `Console`.

```
dcl A fixed bin(15,10);
A = 1.23;

dcl B fixed dec(4,3);
B = 1.230 - A;
```

```
dcl A fixed bin(15,10);
A = 1.23;

dcl B fixed dec(4,3);
B = 1.230 - A;
```

B is 0.003!

```
4/3*pi*r**3
```

- Does this compute the volume?

```
4/3*pi*r**3
```

- Does this compute the volume?

```
4.0/3.0*pi*r**3
```

/ is overloaded.

```
4*pi*r**3/3
```

```
REAL(4)*pi*REAL(r**3)/REAL(3)
```

Why not coerce this?

```
double f(float a, double b) {
  return a*a + b;
}
```

```
double f(float a, double b) {
  return (double)a*(double)a + b;
}
```

Why coerce this?

```
Sub addStrings(a As String, b As String)
  Console.Write(a+b+1)
  Console.Write(1+a+b)
End Sub
```

- What does `addStrings(3, 20)` do?

```
Sub addStrings(a As String, b As String)
   Console.Write(a+b+1)
   Console.Write(1+a+b)
End Sub
```

- What does `addStrings(3, 20)` do?

Writes: `321`
`24`

```
public class Test {
   static void main(String[] args) {
      System.out.println(3+20+1);
      System.out.println(1+3+20);
      System.out.println(1+3+"20");
      System.out.println(1+"3"+20);
   }
}
```

Writes:

```
24
24
420
1320
```

```
public class Test {
   static void main(String[] args) {
      int five = 5;
      Number n = five;
   }
}
```

coerced to `Integer`, which extends `Number`

```
class Int {
  implicit operator Int(int i) {...}
}
class Test {
  static void test(Int z) {
    Console.WriteLine("Int");
  }
  static void test(long l) {
    Console.WriteLine("long");
  }
  static void Main() {
    int i = 5;
    test(i);
  }
}
```

```
class Int {
  implicit operator Int(int i) {...}
}
class Test {
  static void test(Int z) {
    Console.WriteLine("Int");
  }
  static void test(long l) {
    Console.WriteLine("long");
  }
  static void Main() {
    int i = 5;
    test(i);
  }
}
```

should be `public static implicit ...`

```
class Int {
  implicit operator Int(int i) {...}
}
class Test {
  static void test(Int z) {
    Console.WriteLine("Int");
  }
  static void test(long l) {
    Console.WriteLine("long");
  }
  static void Main() {
    int i = 5;
    test(i);
  }
}
```

`int` coerces to `Int`

`int` coerces to `long`

```
class Int {
  implicit operator Int(int i) {...}
}
class Test {
  static void test(Int z) {
    Console.WriteLine("Int");
  }
  static void test(long l) {
    Console.WriteLine("long");
  }
  static void Main() {
    int i = 5;
    test(i);
  }
}
```

int coerces to Int

int coerces to long

error: ambiguous call

```
class List {...}
class OList : List {
  implicit operator BinTree(OList l) {...}
}
class BinTree {
  implicit operator List(BinTree t) {...}
}
class Test {
  static void print(BinTree t) {
    Console.WriteLine("BinTree");
  }
  static void print(List l) {
    Console.WriteLine("List");
  }
  static void Main() {
    print(new OList());
  }
}
```

```
class List {...}
class OList : List {
  implicit operator BinTree(OList l) {...}
}
class BinTree {
  implicit operator List(BinTree t) {...}
}
class Test {
  static void print(BinTree t) {
    Console.WriteLine("BinTree");
  }
  static void print(List l) {
    Console.WriteLine("List");
  }
  static void Main() {
    print(new OList());
  }
}
```

extends `List`, coerces to `BinTree`

coerces to `List`

```
class List {...}
class OList : List {
  implicit operator BinTree(OList l) {...}
}
class BinTree {
  implicit operator List(BinTree t) {...}
}
class Test {
  static void print(BinTree t) {
    Console.WriteLine("BinTree");
  }
  static void print(List l) {
    Console.WriteLine("List");
  }
  static void Main() {
    print(new OList());
  }
}
```

extends `List`, coerces to `BinTree`

coerces to `List`

prints `BinTree`

Pitfalls with coercion

- Implicit computation
- Weaker type checking
- Object identity not preserved
- Value not preserved
- Multistep conversion
- Interaction with overloading
- Interaction with subtyping
- User-defined coercion

Pitfalls with coercion

- Implicit computation
- Weaker type checking
- Object identity not preserved
- Value not preserved
- Multistep conversion
- Interaction with overloading
- Interaction with subtyping
- User-defined coercion

coercion is a type relation; maintain “homomorphism”

prefer subtyping; determine statically

Pitfalls with coercion

- Implicit computation
- Weaker type checking
- Object identity not preserved
- Value not preserved
- Multistep conversion
- Interaction with overloading
- Interaction with subtyping
- User-defined coercion

many subtle issues;
use sparingly

Fortress

- Language for scientific computing
- Mathematical syntax
- Growable
  - everything in libraries (when possible)
  - rich type system
  - very primitive basic types
- Support for parallelism/data distribution

Fortress syntax

```
REAL(4)/REAL(3)*pi*r**3
```

$$\frac{4}{3}\pi r^3$$

Fortress syntax

```
REAL(4)/REAL(3)*pi*r**3
```

$$\frac{4}{3}\pi r^3$$

Basic numeric types are in libraries:
Need user-defined coercion.

Fortress type system

- Trait-based type system
  - supports multiple inheritance of code
  - no inheritance from object trait types
  - traits can exclude other traits
    - object trait types exclude all types other than supertypes
- Overloading for functions/methods
  - symmetric multiple dispatch
  - ambiguous calls prohibited at definition site

Fortress type system

```
trait A
  f(x: A) = “aa”
  f(x: B) = “ab”
end
trait B extends A
  f(x: A) = “ba”
end
```

a.f(a) = “aa”

a.f(b) = “ab”

b.f(a) = “ba”

Fortress type system

```
trait A
  f(x: A) = “aa”
  f(x: B) = “ab”
end
trait B extends A
  f(x: A) = “ba”
end
```

a.f(a) = “aa”

a.f(b) = “ab”

b.f(a) = “ba”

b.f(b) = ??

Fortress type system

```
trait A
  f(x: A) = “aa”
  f(x: B) = “ab”
end
trait B extends A
  f(x: A) = “ba”
end
```

a.f(a) = “aa”

a.f(b) = “ab”

b.f(a) = “ba”

b.f(b) = ??

Fortress forbids this kind of ambiguity:
Must provide “disambiguating” declarations

Fortress type system

```
trait A
  f(x: A) = “aa”
  f(x: B) = “ab”
end
trait B extends A
  f(x: A) = “ba”
  f(x: B) = “bb”
end
```

a.f(a) = “aa”

a.f(b) = “ab”

b.f(a) = “ba”

b.f(b) = “bb”

Fortress forbids this kind of ambiguity:
Must provide “disambiguating” declarations

Overloading

- Exclusion rule
  - > okay if any parameter types exclude each other
- Subtyping rule
  - > okay if all of one's parameter types are “more specific”
- Meet rule
  - > okay if “disambiguating declaration” is provided

Overloading

- Exclusion rule
  - > okay if any parameter types exclude each other
- Subtyping rule
  - > okay if all of one's parameter types are “more specific”
- Meet rule
  - > okay if “disambiguating declaration” is provided

Coercion makes these rules more complicated!

Coercion in Fortress

- Defined in target type
  - must exclude source type
  - target type is less specific
  - not inherited
- Statically determine target type
- No cycles in subtyping/coercion relation
  - “more specific” is (partially) well-defined
- Revised exclusion and meet rules
- Widest-need evaluation

Coercion in Fortress

```
trait A end
object B(x: N) extends A end
object C(x: N)
  coerce(b: B) = C(0)
end
object D(x: N)
  coerce(a: A) = D(1)
  coerce(b: B) = D(2)
  coerce(c: C) = D(3)
end
```

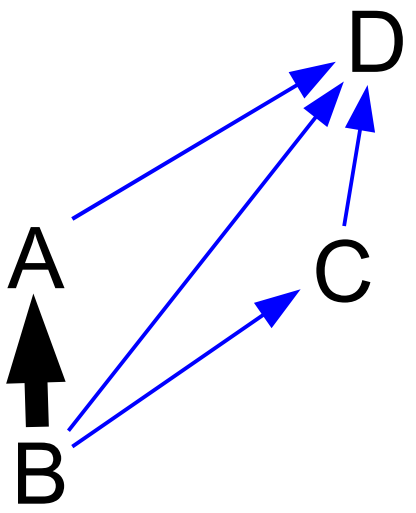
```
object Tester(a: A)
  f(c: C): N = c.x
  f(d: D): N = d.x + 5
  run() = self.f(a)
end
```

What is Tester(B(6)).run()?

Coercion in Fortress

```
trait A end
object B(x: N) extends A end
object C(x: N)
  coerce(b: B) = C(0)
end
object D(x: N)
  coerce(a: A) = D(1)
  coerce(b: B) = D(2)
  coerce(c: C) = D(3)
end
```

```
object Tester(a: A)
  f(c: C): N = c.x
  f(d: D): N = d.x + 5
  run() = self.f(a)
end
```

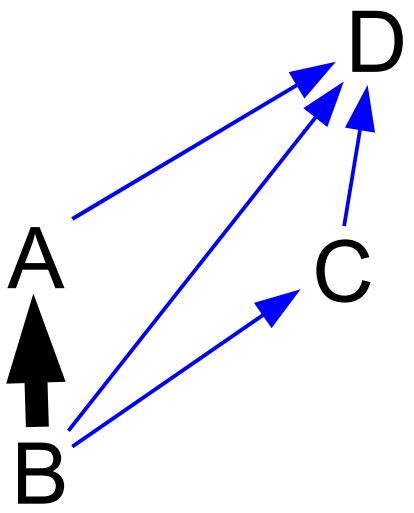


What is Tester(B(6)).run()?

Coercion in Fortress

```
trait A end
object B(x: N) extends A end
object C(x: N)
  coerce(b: B) = C(0)
end
object D(x: N)
  coerce(a: A) = D(1)
  coerce(b: B) = D(2)
  coerce(c: C) = D(3)
end
```

```
object Tester(a: A)
  f(c: C): N = c.x
  f(d: D): N = d.x + 5
  run() = self.f(a)
end
```



What is Tester(B(6)).run()? 7

Overloading with coercion

- Exclusion rule
  - okay if any parameter types exclude each other and no coercion between types
- Subtyping rule
  - okay if all of one's parameter types are “more specific” (by subtyping)
- Meet rule
  - okay if “disambiguating declaration” is provided (now must disambiguate coercions)

Conclusion

- Coercion should be used, but sparingly
  - > subtyping and overloading preferable
- Fortress has user-defined coercion
  - > strong restrictions (enforce no ambiguity)
  - > surprising uses

Come play with us:

http://research.sun.com/projects/plrg

http:///projectfortress.sun.com