Integrating Coercion with Subtyping and Multiple Dispatch

J.J. Hallett
Boston University
jhallett@cs.bu.edu

Victor Luchangco Sukyoung Ryu Guy L. Steele Jr.
Sun Microsystems
{victor.luchangco,sukyoung.ryu,guy.steele}@sun.com

Abstract

Coercion can greatly improve the readability of programs, especially in arithmetic expressions. However, coercion interacts with other features of programming languages, particularly subtyping and overloaded functions and operators, in ways that can produce surprising behavior. We study examples of such surprising behavior in existing languages. This study informs the design of the coercion mechanism of Fortress, an object-oriented language with multiple dynamic dispatch, multiple inheritance and user-defined coercion. We describe this design and show how its restrictions on overloaded declarations prevent ambiguous calls due to coercion.

1. Introduction

Values of different types may represent conceptually distinct entities—the integer `1` and the character `'a'`, for example—or they may be different representations of the same conceptual entity, as with the integer `1` and the floating-point number `1.0`. In the latter case, it is sometimes convenient to use a value of one type where a value of the other type is expected. For example, we may wish to apply a + operator defined on floats to integer arguments. This requires the integers to be *converted* to floats; that is, for each integer argument, we must "compute" a float that represents the same number. Explicitly writing this conversion clutters the code, decreasing readability. Compare, for example, `4/3*pi*cube(r)` to `int2float(4)/int2float(3)*pi*cube(r)`. Thus, many languages [1, 2, 3, 4, 9, 8, 12] provide support for doing this conversion implicitly. This implicit conversion is called *coercion*.

Although coercion improves readability, it can make reasoning about programs more difficult because it introduces implicit computation: a programmer must determine which values must be coerced, and to which type, without any explicit indication in the program text. This difficulty is compounded for languages that support subtyping—especially with multiple inheritance—or overloaded functions or operators, or that allow programmers to define new types that can be coerced to or from existing types. In Section 2, we study examples of the problems that undisciplined use of coercion may cause. This study informs the design of the coercion mechanism of Fortress [5], an object-oriented language for scientific computing that has all the features mentioned above. We describe this design in the context of a stripped-down version of Fortress, and show how its restrictions on overloaded declarations prevent ambiguous calls due to coercion.

SAC'08 March 16-20, 2008, Fortaleza, Ceará, Brazil
ACM 978-1-59593-753-7/08/0003.

2. Problems with Coercion

Because coercion implicitly converts values of one type to values of another, it may mask errors that would otherwise have been caught by type checking, as in the following Visual Basic (versions 4.0 or later) example, adapted from Peterson [10]:

```
Sub printString(str As String, num As Integer)
   For i As Integer = 1 To num
     Debug.Print(str)
   Next i
End Sub
```

This function takes a string and an integer and prints the string the number of times specified by the integer. Because integers may be coerced to strings and vice versa in Visual Basic, a programmer who, intending to call `printString("4",7)`, mistakenly reverses the arguments, calling `printString(7,"4")` instead, is not warned of the error; instead, the arguments are coerced to the expected types, resulting in `"7"` being printed four times.

Coercion can be especially surprising when it occurs on arguments of overloaded functions or operators. For example, we can compute the volume of a sphere in C by evaluating the expression `4*pi*pow(r,3)/3` (where `pi` is a floating-point approximation of π). However, evaluating `4/3*pi*pow(r,3)` yields an *incorrect* answer because `4/3` is an operation on two `int`s, which evaluates to `1`. Thus, C programmers must be careful to distinguish `int`s and `float`s in expressions that involve the `/` operator.

Similarly, consider the following function (also adapted from Peterson [10]) in Visual Basic, which overloads the + operator to add integers and concatenate strings:

```
Sub addStrings(a As String, b As String)
    Debug.Write("a + b + 1 = ")
    Debug.Print(a + b + 1)
    Debug.Write("1 + a + b = ")
    Debug.Print(1 + a + b)
End Sub
```

The arithmetic expressions are evaluated from left to right, with strings coerced to integers whenever one argument of a + operator is a number. Thus, `addStrings(3, 20)` prints:

```
a + b + 1 = 321
1 + a + b = 24
```

This example also raises an important question to consider in understanding coercion: Why are strings coerced to integers and not integers to strings (as in the Java™ programming language, for example)? More generally, when there are two or more ways to convert, or not convert, arguments to yield a valid function or operator invocation, which way is chosen?

This question is especially important in languages with subtype polymorphism, and in which coercion can be defined by the programmer. For example, in C#, values of one class can be coerced to values of another class if either class declares an *implicit conver-*