

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, $4/3 * \pi * \text{cube}(r)$ to $\text{int2float}(4) / \text{int2float}(3) * \pi * \text{cube}(r)$. Thus, many languages [1, 2, 4, 3, 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.

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 $4 * \pi * \text{pow}(r, 3) / 3$ (where π is a floating-point approximation of π). However, evaluating $4/3 * \pi * \text{pow}(r, 3)$ yields an *incorrect* answer because $4/3$ is an operation on two ints, which evaluates to 1. Thus, C programmers must be careful to distinguish ints and floats 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 sub-type 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 conversion operator* from the first class to the second. When resolving a call to an overloaded method, such conversion operators are considered. Indeed, a value whose type is a subtype of the expected type must be “converted” to the expected supertype. Such

conversion is considered on par with any other kind of conversion, which can lead to surprising results, as, for example, in the following C# code:

```
using System;
class A {}
class SubA : A {
    public static implicit operator B(SubA a) {
        return new B();
    }
}
class B {
    public static implicit operator A(B b) {
        return new A();
    }
}
class Test {
    static String announce(B b) {
        return "got a B";
    }
    static String announce(A a) {
        return "got an A";
    }
    static void Main() {
        SubA a = new SubA();
        Console.WriteLine(announce(a));
    }
}
```

The class `SubA` is a subtype of `A`, and can be coerced to class `B`. Class `B` has no subtype relationship with either `A` or `SubA`, but it can be coerced to class `A`. The static `announce` method in class `Test` is overloaded to take either an `A` or a `B`, and returns a string stating which one it is invoked on. By the rules for resolving overloading in C#, an instance of `SubA` is converted to `B` rather than `A` (because `B` coerces to `A`, but not vice versa), even though the latter would not require any change in the underlying representation. Thus, the program prints "got a B", which we believe is surprising to most programmers.

The examples above are surprising because an expression may evaluate to different values depending on its context (e.g., the argument `a` of `addStrings` above may evaluate to an integer or a string), and the effect of applying an operator (e.g., `+`) may be very different for these different values (e.g., integer addition vs. string concatenation). If coercion is intended to preserve the conceptual identity of a value, then we might expect conversion to commute with any operator that can be applied to the value both before and after conversion. For example, we might expect `int2float(4)/int2float(3)` and `int2float(4/3)` to evaluate to the same value.

Even when the target type—the type a value is coerced to—is clear, coercion can be surprising if the conversion procedure is complicated. For example, conversion from one type to another in PL/I [3] may involve a sequence of conversions to intermediate types, a technique we call *chaining*. Consider the following PL/I code, which declares the variable `A` to be a fixed-point binary number with precision attribute $(15, 10)$ —that is, with 15 significant bits, 10 of which are to the right of the radix point—and assigns it the (decimal) value 1.23:

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

The assignment is evaluated by interpreting the character sequence 1.23 as a fixed-point *decimal* number with precision attribute $(3, 2)$ (i.e., 3 significant digits, 2 to the right of the decimal point), converting it to a fixed-point *binary* number with precision attribute $(11, 7)$ —the most nearly equivalent precision attribute in binary to the decimal precision attribute $(3, 2)$ —and then converting that number to one with precision attribute

$(15, 10)$ by “padding” it, resulting in `A` containing the binary number `00001.0011101000`, which represents 1.2265625 in decimal. Thus, this assignment, without any apparent computation, introduces an error of more than 0.003, despite the fact that `A` can contain numbers that are within 0.001 of each other. (We could get more precision by writing 1.230 instead of 1.23.)

3. The Fortress Programming Language

Fortress is a new object-oriented language targeted primarily at scientific computing, but also intended to support general-purpose programming [5]. One design principle of Fortress is that it should be *growable* [11]. In particular, whenever possible, Fortress provides language features through libraries rather than wiring them into the compiler, so that these features can be adapted, and new features added, as the language and the needs of its users evolve. Even “primitive” data types such as Boolean, Integer and Float are defined in libraries.

To support the definition and evolution of such types in libraries, Fortress has a rich parametric trait-based type system. *Traits* [7] are like Java interfaces in that they exist in a multiple inheritance hierarchy, but unlike interfaces, they may contain code (but not fields). *Objects* extend traits and may contain fields as well as code, but cannot be extended. The Fortress type system can express type relations beyond subtyping. In particular, it can express that two types *exclude* each other; that is, that no value is an instance of both types.

Fortress also allows functions, methods and operators to be *overloaded*: a method, for example, may have multiple definitions, each with a different set of parameter types. When the method is invoked, overloading is resolved by *multiple dynamic dispatch*: the definition used is the *most specific applicable* one based on the run-time types of all the arguments. We formalize this overloading resolution rule in the context of a subset of Fortress in the next section. To avoid surprising behavior, Fortress forbids ambiguous calls—ones in which there are multiple applicable definitions but none more specific than all the others. Indeed, Fortress forbids a set of overloaded method definitions if any application of the resulting method may be ambiguous, whether or not any such call is actually made. The rules for legal overloading are also formalized in the next section.

Because basic numeric types such as Integer and Float are defined in libraries, and because Integer should coerce to Float, Fortress supports user-defined coercion. However, coercion should not be introduced lightly. Rather, allowing coercion from type `A` to type `B` should be a deliberate decision by the designer of `B` that in every circumstance that a function, method or operator expects an argument of type `B`, a value of type `A` may be used instead. In particular, every instance of `A` should represent the same conceptual entity as some instance of `B`. Thus, the relationship between `B` and `A` is similar to the one between `B` and its subtypes. We say that `A` is *substitutable* for `B` if either `A` is a subtype of `B` or `A` coerces to `B`.

Coercion is defined in Fortress similarly to methods in the declaration of the target type. Like methods, coercion can be overloaded: a target type may define coercion from multiple types, and the definition used is determined by dynamic dispatch. However, a coercion definition differs from methods in several ways: A coercion definition always has a single parameter and does not declare a return type (it is always the type containing the coercion definition). Coercion definitions are not inherited; indeed, it would not be type sound to inherit coercion definitions: the result of coercion to the supertype might not be an instance of the subtype. Also, coercion in Fortress, unlike PL/I, does not chain: a value being coerced must be of the type (or a subtype) of the parame-

$$\begin{aligned}
p & ::= \vec{d} e \\
d & ::= td \mid od \\
td & ::= \text{trait } T \text{ extends } \{\vec{T}\} \text{ excludes } \{\vec{T}\} \vec{cd} \vec{md} \text{ end} \\
od & ::= \text{object } O(\vec{x}:\vec{\tau}) \text{ extends } \{\vec{T}\} \vec{cd} \vec{md} \text{ end} \\
cd & ::= \text{coerce}(x:\tau) = e \\
md & ::= f(\vec{x}:\vec{\tau}):\tau = e \\
e & ::= x \mid \text{self} \mid O(\vec{e}) \mid e.x \mid e.f(\vec{e}) \mid n \\
\tau & ::= T \mid O \mid \mathbb{N}
\end{aligned}$$

Figure 1. Grammar for CFC syntax: The metavariable T ranges over trait names, O over object names, f over method names, x over field and variable names, and n over the natural numbers (type \mathbb{N}). An arrow over a term indicates that the term may occur zero or more times.

ter type specified by a coercion definition; it is not coerced to the parameter type.

To avoid problems described in the previous section, Fortress statically determines whether coercion is required, and if so, what the target type of the coercion is. In Fortress, unlike C#, upcasting is preferred to coercion: a value is coerced only if the expression would otherwise be rejected by static type checking. Thus, it serves no purpose to coerce from a type to any of its supertypes, and Fortress statically forbids such coercion from being defined. In addition, Fortress forbids cycles in the substitutability relation. For example, the following set of declarations is forbidden:

```

trait A end
trait B extends A end
trait C extends B
  coerce (a: A) = ...
end

```

because C is substitutable for B (since C is a subtype of B), B is substitutable for A (B is a subtype of A), and A is substitutable for C (A coerces to C).

4. Overloading Rules

In this section, we formalize the rules for defining overloaded methods and resolving method calls in the presence of coercion; rules for functions and operators are analogous, and we omit them for brevity. These rules statically prevent ambiguous calls. We present these rules in the context of CFC (core Fortress with coercion), a distillation of those aspects of Fortress relevant to integrating coercion with subtyping and method overloading. This builds on previous work [6], which describes rules for defining and resolving overloading that prevent ambiguous calls statically and with modular checks.

The syntax of CFC appears in Figure 1. A program is composed of a sequence of trait and object declarations followed by an expression. The value of the program is the value of the expression evaluated in the context of the declarations.

A trait or object declaration contains coercion and method definitions, and extends zero or more traits. An object also declares fields in its header. A trait explicitly excludes zero or more traits.

A trait or object is a (strict) subtype of the traits it extends and any of their supertypes. A trait excludes the traits it explicitly excludes and any of their subtypes, as well as any traits that exclude it (exclusion is a symmetric relation). No trait or object may extend a trait that excludes it or nor may it extend two traits that exclude each other. Because objects cannot be extended, every object excludes every other object, as well as any trait that is not its supertype.

A trait or object inherits the method definitions of its supertypes. A trait or object *provides* a method if it defines the method or inherits such a definition from a trait it extends. A trait or object may provide multiple definitions for a single method; such definitions are overloaded.

For simplicity, we assume that there is no overlap among the different kinds of names; that for each trait or object name, there is a unique declaration corresponding to it; that field names for a single object are all distinct; and that variable names for a single method or coercion definition are all distinct, and also distinct from all the field names of the enclosing object (if the method/coercion definition is in an object declaration).

To reduce clutter, we omit `extends` and `excludes` clauses when they contain no types.

Given types A and B , we write $A < B$ if A is a strict subtype of B , $A \leq B$ if $A < B$ or $A = B$, and $A \diamond B$ if A excludes B . We also write $A \rightarrow B$ if B defines a coercion from A , and $A \rightsquigarrow B$ if B defines a coercion from A or any supertype of A . We extend this notation to tuples of types in the obvious way. For example, $(A_1, \dots, A_k) \rightarrow (B_1, \dots, B_k)$ if $(A_1, \dots, A_k) \neq (B_1, \dots, B_k)$ and for all i , either $A_i \rightarrow B_i$ or $A_i = B_i$.

4.1 Resolving Overloading

Without coercion, overloading is resolved by dispatch based on the run-time types of the arguments. With coercion, recall that we want to *statically* determine which arguments will be coerced, and to what target types. Thus, overloading with coercion is resolved in two stages: first by determining coercion at compile time, and then by dispatch at run time.

For both determining coercion and dispatch, the first step is to determine the definitions applicable to the call. But the notion of applicability is slightly different for each case: A method definition is (*directly*) *applicable* to a call if it is provided by the receiver's type and the types of the arguments are subtypes of or equal to the corresponding parameter types of the definition. It is *applicable with coercion* under the same condition using substitutability rather than subtyping to compare the argument and parameter types. Recall that A is substitutable for B if $A < B$ or $A \rightsquigarrow B$.

Applicability with coercion is used only to determine coercion at compile time. Thus, it is always based on the static types of the arguments and receiver. In contrast, direct applicability is used for both dispatch and determining coercion: Because we prefer upcasting to coercion, no coercion is mandated if any definition is applicable based on the static types of the arguments and receiver. We say such a definition is *statically (directly) applicable*, whereas whether a definition is *dynamically applicable* is based on the run-time types.

Once the applicable definitions are determined, we choose the most specific one, that is, the one whose receiver and parameter types are more specific than the corresponding types of any other applicable definition. For dispatch, the standard notion based on subtyping is sufficient. But for determining coercion, we need an expanded notion. Informally, we want to say that type A is more specific than type B if A is a subtype of B , or if A coerces to B and neither B nor any subtype of B coerces to or is a subtype of A . However, we cannot always determine all the subtypes of a type (due to modularity constraints, which are beyond the scope of this paper). Therefore, we adopt a more conservative notion, defined precisely below.

For types A and B , we say that A *rejects* B , and write $A \not\rightsquigarrow B$, if for every coercion defined by A , the parameter type of the coercion definition excludes B :

$$A \not\rightsquigarrow B \iff \forall C, C \rightarrow A \implies C \diamond B$$

Note that $A \not\prec B$ implies $\neg(B \prec A)$ and $\forall C, C \prec A \implies C \diamond B$. We say that A is *more specific* than B , and write $A \triangleleft B$, if A is a subtype of B or A excludes, coerces to, and rejects B :

$$A \triangleleft B \iff A < B \vee (A \diamond B \wedge A \rightsquigarrow B \wedge A \not\prec B).$$

Given a nonempty set of types, a type in that set is *maximally specific* if no type in the set is more specific than it. A type is the *most specific* type in a set if it is the only maximally specific one.

We write $A \preceq B$ if $A \triangleleft B$ or $A = B$, and we extend this notation in the obvious way to tuples of types.

Note that \preceq might not be a partial order: it is reflexive and antisymmetric but not necessarily transitive. For example, if each of A , B and C exclude the other two, and A coerces to B and B coerces to C but A does not coerce to C , then $A \preceq B$ and $B \preceq C$ but $A \preceq C$ does *not* hold. Nonetheless, because the substitutability relation is acyclic and a type is more specific than its supertypes, a definition that is most specific by subtyping is also most specific by this notion.

A method call is thus resolved as follows: If any definition is statically applicable then resolve overloading by ordinary dynamic dispatch. Otherwise, determine the most specific definition that is applicable with coercion. (If no definition is applicable with coercion then the method call is undefined, and the program is statically rejected.) Designate each argument whose static type is not a subtype of or equal to the corresponding parameter type of the definition for coercion to the parameter type. At run time, coerce each argument so designated to the appropriate target type using the most specific applicable coercion definition. Then with the resulting arguments, dispatch to the most specific applicable method definition. The rules in the next subsection guarantee that this procedure is well-defined; that is, for any method call for which some definition is applicable, there is a most specific such definition.

Alternatively, imagine that for every type A , there is a special, possibly overloaded function $coerce_A$, defined by the coercion definitions in type A . In addition, the function is the identity function on values of type A . At compile time, determine the most specific statically applicable method definition, or if there is none, the most specific method definition applicable with coercion. Replace each argument a with a call to $coerce_A(a)$, where A is the corresponding parameter type of that definition. At run time, use ordinary dynamic dispatch both for the special coercion functions and for the method call after the coercion is complete.

As an example, consider the following program:

```

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 E(x: N)
  coerce (a: A) = E(4)
  coerce (d: D) = E(5)
end
object Tester(a: A)
  f(c: C): N = c.x
  f(d: D): N = d.x + 7
  f(e: E): N = e.x + 14
  run() = self.f(a)
end
Tester(B(6)).run()

```

This evaluates to 9: No definition is applicable to `self.f(a)`, and the most specific definition applicable with coercion is $f(D)$:

$$f(d: D): \mathbb{N} = d.x + 7$$

because $f(C)$ is not applicable with coercion and $D \triangleleft E$. Thus, a is coerced to D . At run time, $a = B(6)$, so the coercion invoked is:

$$\text{coerce } (b: B) = D(2)$$

4.2 Defining Overloaded Methods

We now give rules for defining valid overloaded methods that ensure that no method call is ambiguous. Each rule applies to pairs of overloaded definitions; a pair that satisfies a rule cannot cause ambiguous calls. Thus, if every pair of overloaded definitions satisfies one of these rules, then no ambiguous calls are possible. That is, if \mathcal{S} is the set of overloaded method definitions provided by type X for method f , then \mathcal{S} is a *valid overloading* if each pair of distinct definitions in \mathcal{S} satisfies the subtype rule, the exclusion rule, or the meet rule with \mathcal{S} , as defined below. These rules are adapted from the rules of [6] to account for coercion.

To concisely state these rules, we use $P_0.f(P): P_r$ to denote a definition of f , where P_0 is the trait or object type in which the definition appears, P is the tuple of parameter types, and P_r is the return type. Such a designation uniquely identifies a definition because a trait or object must not contain two method definitions with the same name and parameter types.

The Subtype Rule $P_0.f(P): P_r$ and $Q_0.f(Q): Q_r$ satisfy the subtype rule if $(P_0, P) < (Q_0, Q)$ and $P_r \leq Q_r$.

The rule above is unchanged by coercion, because it depends only on subtyping. A pair of definitions that satisfies the subtype rule never introduces ambiguity, because one of them is always more specific than the other.

The Exclusion Rule $P_0.f(P): P_r$ and $Q_0.f(Q): Q_r$ satisfy the exclusion rule if $P \diamond Q$ and $P \not\prec Q$ and $Q \not\prec P$ and for all A and B , $A \rightarrow P \wedge B \rightarrow Q \implies A \diamond B$.

Two definitions that satisfy the exclusion rule never introduce ambiguity because there is no set of arguments to which both definitions are applicable, even with coercion. Coercion makes this rule harder to satisfy because it makes definitions applicable to more calls. We require not only that no value is an instance of both types, but also that no value can be coerced to both types, and that no instance of one type can be coerced to the other.

The meet rule is trickier because it depends not only on the two definitions, but also on the set of all provided definitions of a method. To write this rule, we first introduce some terminology: For types A and B , let $A \cap B$ be A if $A \leq B$, B if $B < A$, and undefined otherwise. We extend this elementwise to tuples of types. Given a set \mathcal{S}_f of definitions for a method f , we say that two pairs (A_0, A) and (B_0, B) , each of a type and a tuple of types, are *disambiguated in \mathcal{S}_f* if either $A \diamond B$ or there is a definition $R_0.f(A \cap B): R_r$ in \mathcal{S}_f with $R_0 \leq A_0$ and $R_0 \leq B_0$. If (A_0, A) and (B_0, B) represent the receiver and parameter types of method definitions, then disambiguation in \mathcal{S}_f is essentially the meet rule without coercion. However, the meet rule is generalized here to handle the case in which the arguments of a method call may be coerced to A and/or B .

The Meet Rule If \mathcal{S}_f is a set of overloaded method definitions, then $P_0.f(P): P_r$ and $Q_0.f(Q): Q_r$ satisfy the meet rule with \mathcal{S}_f if both of the following conditions hold:

- (P_0, P) and (Q_0, Q) are disambiguated in \mathcal{S}_f ; and
- either $P \triangleleft Q$ or $Q \triangleleft P$ or for all $A \rightarrow P$ and $B \rightarrow Q$, (P_0, A) and (Q_0, B) are disambiguated in \mathcal{S}_f .

If two definitions provided by a trait satisfy the meet rule with a set of definitions also provided by the trait, then they do not introduce ambiguity because:

- (by the first condition) either (1) there is no method call to which both are directly applicable, or (2) the trait provides some definition that is more specific than or equal to each and is applicable whenever both are applicable, and
- (by the second condition) if both are applicable with coercion (but not directly applicable) to a method call, then either (1) one is more specific than the other, or (2) the trait provides some definition that is more specific than both and is applicable with coercion whenever both are applicable with coercion.

Thus, for any method call for which it is ambiguous which of the two definitions would be better to use, there is another declaration that is better than both.

Note that these rules are for overloaded method definitions, not for overloaded coercion definitions. The rules for coercion definitions are simpler, like those in [6], because the “arguments” of coercion must not have already been coerced (since coercion doesn’t chain in Fortress).

To show that restrictions on overloaded definitions prevent ambiguity, we must show that both coercion resolution at compile time and dispatch at run time are well defined. We prove the former in Appendix A; the latter follows with minor modifications from [6]. Together these guarantee that there are no undefined nor ambiguous calls at run time.

5. Conclusion

In this paper, we present a new design for supporting user-defined coercion in an object-oriented language with support for overloaded methods and multiple dispatch. This design is informed by our study of problems introduced by coercion in previous languages. In particular, we differentiate coercion from method dispatch, statically determining where coercion occurs, while still allowing dispatch to be based on the run-time types of a method’s arguments. We give restrictions on how coercion and overloaded methods are defined and prove that these restrictions prevent ambiguous calls at run time.

Acknowledgments

We thank Eric Allen, David Chase and Jan-Willem Maessen for feedback on the ideas in this paper.

References

- [1] Fortran 90, ISO/IEC 1539: 1991 (E), ANSI X3.198-1992, May 1991.
- [2] C++, ISO/IEC 14882: 1998, 1998.
- [3] Enterprise PL/I for z/OS: Enterprise PL/I Language Reference, Seventh Edition, November 2005.
- [4] C# Language Specification, 4th Edition, ECMA-334, June 2006.
- [5] Eric Allen, David Chase, Joe Hallett, Victor Luchangco, Jan-Willem Maessen, Sukyoung Ryu, Guy L. Steele, Jr., and Sam Tobin-Hochstadt. The Fortress Language Specification Version 1.0 beta, March 2007.
- [6] Eric Allen, J. J. Hallett, Victor Luchangco, Sukyoung Ryu, and Guy L. Steele, Jr. Modular Multiple Dispatch with Multiple Inheritance. In *Object-oriented Programming Languages and Systems, Special Track at the 22nd Symposium on Applied Computing, SAC-07*, Seoul, Korea, March 11-15, 2007. ACM.

- [7] Stéphane Ducasse, Oscar Nierstrasz, Nathanael Schärli, Roel Wuyts, and Andrew P. Black. Traits: A mechanism for fine-grained reuse. *ACM Trans. Program. Lang. Syst.*, 28(2):331–388, 2006.
- [8] James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *The Java Language Specification, Third Edition*. 2005.
- [9] B. W. Kernighan and D. M. Ritchie. *The C Programming Language, Second Edition*. Prentice-Hall, Englewood Cliffs, New Jersey, 1988.
- [10] Karl E. Peterson. Coercion aversion. *Visual Basic Programmer’s Journal*, pages 148–154, November 1995.
- [11] Guy L. Steele, Jr. Growing a language. Keynote talk, OOPSLA, 1998. Also published at *Higher-Order and Symbolic Computation* 12, 221–236, 1999.
- [12] Paul Vick. The Microsoft Visual Basic Language Specification, Version 8.0 (Beta 2), 2005.

A. Proof of Coercion Resolution for Methods

We prove that the overloading rules guarantee the static resolution of coercion is well defined for methods. We use *static call* $A_0.f(A)$ to refer to a call with name f whose receiver and argument have static types A_0 and A , respectively.

Consider a static call $A_0.f(A)$. Let Σ be the set of parameter types of method declarations of f that are provided by A_0 and directly applicable to the static call $A_0.f(A)$. Let Σ' be the set of parameter types of method declarations of f that are provided by A_0 and applicable with coercion to the static call $A_0.f(A)$. Moreover, let σ' be the subset of maximally specific types of Σ' :

$$\sigma' = \{S \in \Sigma' \mid \neg \exists S' \in \Sigma' : S' \triangleleft S\}.$$

We prove $|\Sigma| = 0$ and $|\Sigma'| \neq 0$ imply $|\sigma'| = 1$. That is, if no declaration is applicable to a static call but some declaration is applicable with coercion then there is a most specific declaration that is applicable with coercion.

Lemma 1. *If $|\Sigma'| \geq 1$ then $|\sigma'| \geq 1$.*

Proof. Consider the graph of the relation \triangleleft on the types in Σ' . Then σ' is the set of nodes in this graph with no in-edges. Because \triangleleft is acyclic and irreflexive and Σ' is finite, there is at least one such node, so $|\sigma'| \geq 1$. \square

Lemma 2. *If $|\Sigma| = 0$ then $|\sigma'| \leq 1$.*

Proof. Suppose P and Q are two types in Σ' such that neither $P \triangleleft Q$ nor $Q \triangleleft P$ holds. We show that neither P nor Q is maximally specific in Σ' . Thus, there cannot be two maximally specific types in Σ' , so $|\sigma'| \leq 1$, as required.

By the definition of Σ' , A_0 provides declarations $A_0.f(P)$ and $A_0.f(Q)$ that are applicable by coercion to $A_0.f(A)$. These declarations must satisfy one of the overloading rules from Section 4.2. Because neither $P \triangleleft Q$ nor $Q \triangleleft P$ holds, they cannot satisfy the subtype rule. Because they are applicable with coercion to $A_0.f(A)$, but not directly applicable (since $|\Sigma| = 0$), there must exist types P' and Q' such that $A \leq P' \rightarrow P$ and $A \leq Q' \rightarrow Q$, so $\neg(P' \diamond Q')$. Therefore they cannot satisfy the exclusion rule. Thus, they must satisfy the meet rule. Since neither $P \triangleleft Q$ nor $Q \triangleleft P$ holds, and $\neg(P' \diamond Q')$, then by the second condition of the meet rule applied to P' and Q' , there is a declaration $f(P' \cap Q')$ provided by A_0 . This declaration is applicable with coercion to $A_0.f(A)$, so $P' \cap Q' \in \Sigma'$. Thus, neither P nor Q is maximally specific in Σ' (since $P' \cap Q'$ is more specific than each of them). \square

Theorem 1. *If $|\Sigma| = 0$ and $|\Sigma'| \neq 0$ then $|\sigma'| = 1$.*

Proof. Immediate from Lemmas 1 and 2. \square