© APR 2026 | IRE Journals | Volume 9 Issue 10 | ISSN: 2456-8880
DOI: https://doi.org/10.64388/IREV9I10-1717077

Formal Analysis of Overloaded Subprogram Resolution Rules: Ambiguity, Coercion, and Default Parameters in C++, Java, C#, and Ada

AIBUKI ADEYINKA[1], OWOLABI HAMMED[2], OGOZI CONFIDENCE[3], ADEWOLE PHILIPS[4]
[1, 2, 3, 4]*Department of Computer Science, University of Lagos, Nigeria*

Abstract- ***A simple feature in the design of a statically typed programming language is subprogram overloading: the same identifier can have a variety of implementations in which the arguments have different types. But when implicit type coercion and default parameters are used in conjunction with overloading, the resolution is much more complicated and may result in ambiguities not automatically resolved by the compiler. This paper provides a formal comparative study of overloaded subprogram resolution in four popular languages: C++, Java, C# and Ada. It compares the language-specific mechanisms such as conversion-rank hierarchy in C++, three-phase method invocation procedure in Java, optional parameter and user-defined conversion rules in C#, and top-down expected-type propagation in Ada . The paper shows that none of the four languages can ensure SNA when implicit coercions of equal rank interact with overloaded candidates at the same time, and suggests design principles such as total-order ranking function and default-conscious candidate selection, which would restore SNA in such situations.***

Index Terms- ***Overload Resolution, Type Coercion, Ambiguity, Default Parameters, C++, Java, C#, Ada***

I. INTRODUCTION

The assignment of the same name to two or more subprograms in the same scope is known as overloading; the correct implementation will be selected at compile time, depending on the types of the arguments provided. This facilitates a programming style that is ad hoc polymorphism, where a single operation name can be used on arguments of various types (Scott, 2016). Although the concept is generally well-known, the rules that govern the process by which compilers do this selection, which is often known as overloaded subprogram resolution, differ significantly across programming languages, and are even more complicated when two other features are involved: implicit type coercion and default parameter values.

Implicit coercion permits one type of value to be implicitly converted to another type, when it is needed by the calling context. Default parameters enable a caller to skip one or more arguments which the declaration provides with default values. Both of these features alone contribute to the expressiveness of a language, by increasing the set of valid calls. But every feature has an independent effect of multiplying the number of candidate subprograms which can be applied to any one particular call site. The interactions between both features when combined with overloading allow the enlarged candidate set to contain two or more equally ranked candidates, an ambiguity that cannot be automatically resolved by any rule in the language.

The main research problem of the paper is to understand the time and reason when this occurs, and how it is addressed by each of the languages.

Problem Statement: The paper explores the three-way interplay of subprogram overloading, implicit type coercion, and default parameter values to create ambiguities in C++, Java, C#, and Ada; formally describes the resolution strategy used by each language; and assesses whether the resolution strategies meet the Strong Non-Ambiguity (SNA) condition.

The discussion is constructed on common vocabulary. The coercion relation (written delta) refers to the space of available implicit conversions in a particular language. The ranking operation (denoted rho) is a preference ranking of competing paths of coercion. The type context (written as Gamma) provides the information of the calling environment about expected-type. C++ and Java assign Gamma to empty, i.e. they are bottom-up resolvers which do not access contextual type information. Ada fills up Gamma by its top-down propagation process. Such parameters are employed in Sections 3.2–3.5 to describe each language and to determine the precise circumstances in which SNA stops working.

The rest of this paper is structured as follows. Section 3.1 presents a background of formal overloading theory, and the SNA criterion. The language analysis of C++, Java, C# and Ada (Sections 3.2–3.5 respectively) is done with formal rules and worked code examples. Section 4.1 has a comparative analysis. Section 4.2 discusses design implications and Section 5 concludes.

III. RESEARCH ELABORATIONS

3.1 BACKGROUND: FORMAL OVERLOADING THEORY

3.1.1 Type-Directed Overloading

Formally, the overloaded subprogram resolution can be viewed as a type-directed rewriting. In a compiler, an occurrence of a call f(e1, e2,..., en) would have to pick precisely one of the visible declarations which bear the name f. This choice is determined by comparing the types of arguments to the types of parameters of each declaration, possibly with implicit coercions to overcome type mismatches. A coercion set is defined as a collection of typed functions, which each take a value of one type as an input and a value of another type as an output, and a coercion generation process that inserts the elements of this set into a source term to render it type-correct. It is sound when it always yields a well-typed result, and it is Strong Non-Ambiguous when it yields just a single well-typed result (Swamy et al., 2009).

3.1.2 Strong Non-Ambiguity (SNA)

Strong Non-Ambiguity, as defined by Swamy et al. (2009), is the formal specification that there is a unique coercion rewriting for every program expression. SNA is true for a resolution system if and only if the following three conditions are true.

The first is NAC-pi, which states that there are unique coercion paths: between any two types T1 and T2 there should be at most one coercion path of a given length in the coercion graph. The second is NAC-times, which requires unique coercion to a product type: when a number of arguments are considered as a tuple, there must be a unique coercion path to any given product type. Finally, the third is NAC-app, which requires unique applications: at each function application site, there must be a unique path to a function type. If any of these conditions is violated then the call is ambiguous and the compiler must raise an error (Swamy et al., 2009).

SNA is a valuable line of analysis since it is independent of the resolution algorithm used by a particular language. A language either complies with it (for a program) or not, regardless of the steps the compiler takes to determine the violation. As we show in the next sections, SNA is violated by all four languages for some cases of equal-rank coercions.

3.1.3 Coercion Categories

Types of coercions involved in overloading are classified into two (Luo, 2008). Argument coercions transform an actual argument to the type of a formal argument. Function coercions convert the return type of the subprogram at the call site. These are present in all the languages analysed here, albeit to different extents. C++ and C# have user-defined argument coercions (conversion operators); Java has only built-in widening conversions; Ada has built-in conversions for numeric types with top-down type propagation.

The interaction of argument coercions and subtyping is the main source of Type-II ambiguities, where two different candidates are ranked equally by the ranking function. (Allen et al., 2008) studied this interaction in a number of programming languages and demonstrated that it leads to surprising results if the coercion relation is not endowed with a total ordering.

3.2 OVERLOAD RESOLUTION IN C++

3.2.1 Candidate Set Algorithm

The C++ overload resolution process is defined in ISO/IEC 14882 and is a three-step process. First, the compiler creates the set of candidates: all declarations visible at the call site with the same name, including function templates after type deduction. Second, it narrows to viable candidates: those with the correct number of parameters and at least one implicit conversion sequence for each argument. Third, it picks the best viable candidate by ranking the implicit conversion sequence for each argument position (ISO/IEC, 2020).

Conversion sequences come in four ranks: (1) exact match or const qualification difference, (2) promotions (e.g. short to int, float to double), (3) standard conversions (e.g. int to double), and (4) user-defined conversions. Any promotion is preferred over any standard conversion, and any

© APR 2026 | IRE Journals | Volume 9 Issue 10 | ISSN: 2456-8880
DOI: https://doi.org/10.64388/IREV9I10-1717077

standard conversion over any user-defined conversion (ISO/IEC, 2020).

3.2.2 Coercion and Ambiguity: A Worked Example
The ranking hierarchy resolves many potential conflicts, but does not define a total order within tiers. When two candidates each require a conversion from the same tier, neither dominates, and the call is ambiguous:

```
void transform(long x);
void transform(float x);
transform(42); // argument type:
int
// int to long:  promotion (tier
2)
// int to float: promotion (tier
2)
// Equal rank: compile-time
ambiguity error
```

Both conversions from int to long and int to float are promotions and thus at the same tier. The SNA condition NAC-app fails: there are two equally good coercions to two different candidates, so the compiler gives an error (ISO/IEC, 2020).

3.2.3 C++ Defaults and Ambiguity
C++ allows default values for function arguments, enabling a function declaration to be matched by calls with different numbers of arguments. This leads to ambiguity if two declarations have identical effective parameter profiles for a given argument count:

```
void log(int code, int level,
int severity = 0);
void log(int code, int level);
log(5, 2);
// Both match exactly on two int
arguments.
// Compile-time ambiguity error.
```

The C++ tie-breaking rules (ISO/IEC, 2020) do not prefer the function without the default over the one with it because both are an exact match. Default-induced ambiguity occurs only when two declarations have an identical effective profile for a particular number of arguments.

3.3 OVERLOAD RESOLUTION IN JAVA

3.3.1 Method Invocation in Three Phases
The method overloading resolution in Java is defined in Section 15.12 of the Java Language Specification (Gosling et al., 2023) and is done in three phases ordered to ensure backward compatibility. Phase 1 finds applicable methods using subtyping and widening conversions, but not boxing, unboxing, or variable-arity invocation. If one or more applicable methods are found by Phase 1, Phases 2 and 3 are bypassed. Phase 2 uses boxing and unboxing. Phase 3 allows variable-arity invocation (Gosling et al., 2023).

If there is more than one applicable overload, Java selects the most specific method. A method m1 is more specific than m2 if the argument types of m1 are subtypes of the argument types of m2. If two methods are mutually non-specific, an ambiguity error occurs (Gosling et al., 2023).

3.3.2 Java's Coercion and Ambiguity
There are no user-defined implicit conversions in Java. The coercion relation consists only of built-in widening primitive conversions and widening reference conversions. The most-specific rule is based on subtyping, not conversion cost: all widening primitive conversions are considered equally applicable. This results in Type-II ambiguities:

```
void compute(long x)  { ... }
void compute(float x) { ... }
compute(5);
// int widens to both long and
float
// long and float not comparable
by subtyping
// AMBIGUOUS
```

3.3.3 Default Parameters: Not a Feature
The Java Language Specification does not provide for default parameters. Optional arguments can be supported only by programmers explicitly declaring overloads. This choice removes Type-III ambiguities from Java altogether: since there is no default-expansion step, the candidate set input to the resolution algorithm is identical to the set of programmer declarations, simplifying resolution (Gosling et al., 2023).

3.4 OVERLOAD RESOLUTION IN C#

3.4.1 Applicable Candidates and User-Defined Conversions

The details of C# overload resolution can be found in ECMA-334 (ECMA International, 2022). The overall framework is similar to Java: find applicable candidates, then choose the best one. Two major extensions create additional ambiguity potential: (1) programmers can declare user-defined implicit conversion operators using the implicit operator keyword; (2) programmers can declare optional parameters with default values and named arguments.

ECMA-334 (ECMA International, 2022) does not specify a preference between user-defined implicit conversions that share a common source type. If there are two user-defined conversions from S to T1 and T2, and two overloads with parameters of types T1 and T2, neither overload is preferred — a Type-II ambiguity:

```
struct Metres {
    public static implicit
operator double(Metres m)
        => m.Value;
    public static implicit
operator float(Metres m)
        => (float)m.Value;
    public double Value;
}
void measure(double d) { }
void measure(float f)  { }
Metres m = new Metres { Value =
1.5 };
measure(m); // AMBIGUOUS
```

3.4.2 Named Arguments

C# optional parameters interact with overloading via candidate expansion. ECMA-334 provides a partial tie-breaker: if two applicable candidates differ only in the number of optional arguments substituted by their default value, the one requiring fewer substitutions is preferred (ECMA International, 2022). However, when two candidates require the same number of default substitutions and are otherwise tied by conversion, the call remains ambiguous. Note also that the usual C# overload resolution (without the dynamic keyword) is a static, compile-time process.

3.5 OVERLOAD RESOLUTION IN ADA

3.5.1 Top-Down Expected-Type Propagation

Ada's overload resolution algorithm has a different direction to that of C++, Java and C#. Rather than resolving bottom-up, the expected type of each argument expression is found by propagating the expected type top-down from the call to each argument expression. This is described in Section 8.6 of the Annotated Ada Reference Manual (Ada-Auth, 2012). Ada allows overloaded enumeration literals and numeric literals, and the formal parameter type of the call determines what type the literal takes. When the subprogram called and the argument expression are simultaneously overloaded on the same types, the top-down propagation has two equally consistent complete interpretations and requires programmer intervention to disambiguate (Ada-Auth, 2012).

3.5.2 Complete Interpretations and Ambiguity

Resolution is successful if there is one and only one acceptable interpretation: an assignment of all declared entities to all names in the complete context that satisfies all expected-type constraints. If there are two such interpretations, the call is ambiguous and a qualified expression must be used:

```
function Empty return Sequence;
procedure Print (S : Sequence);
function Empty return Set;
procedure Print (S : Set);
Print (Empty);
-- AMBIGUOUS: two consistent
interpretations
Print (Sequence'(Empty));
-- RESOLVED: qualified
expression
```

3.5.3 Ada's Default Parameters

Ada supports defaults for in-mode parameters. Ada's top-down context can eliminate some Type-III ambiguities because the expected type propagated from the calling context can differentiate between two expanded signatures with the same number of arguments. If no disambiguating type is provided by the calling context, the same Type-III ambiguity as in C++ and C# occurs in Ada, and must be resolved by a qualified expression or renaming the declarations.

IV. RESULTS OR FINDING

4.1 COMPARATIVE ANALYSIS

4.1.1 Resolution Strategy Summary

The table below lists the resolution strategy and ambiguity properties of each language, as analysed in Section 3.2 to 3.5.3

© APR 2026 | IRE Journals | Volume 9 Issue 10 | ISSN: 2456-8880
DOI: https://doi.org/10.64388/IREV9I10-1717077

Language	Direction	User Coercions	Default Params	Ambiguity Types
C++	Bottom-up, 4-tier rank	No (built-in)	Yes	Type-I, II, III
Java	Bottom-up, most-specific	No (built-in)	No	Type-I, II only
C#	Bottom-up + named args	Yes (user-defined)	Yes	Type-I, II, III
Ada	Top-down, expected type	No (built-in)	Yes	Type-I, III (reduced)

Table 1: Overload resolution properties across C++, Java, C#, and Ada.

4.1.2 The Triple Interaction: Overloading, Coercion and Defaults

The key insight of this paper is that none of the four languages ensure SNA in the presence of implicit coercions with equal rank competing with overloaded candidates at the same call site. Below is a multi-language example of the same structural ambiguity:

```
// C++ (ISO/IEC, 2020):
void open(int fd,    int mode =
0);
void open(double id, int mode =
0);
open(42L); // long->int & long-
>double both tier 2: AMBIGUOUS

// Java (Gosling et al., 2023):
void open(long fd)  { }
void open(float id) { }
open(42); // int widens to both;
neither subtype: AMBIGUOUS

// C# (ECMA International,
2022):
void open(int fd,    int mode =
0) { }
void open(double id, int mode =
0) { }
open(myFileId); // two equal-
rank user conversions: AMBIGUOUS

// Ada (Ada-Auth, 2012):
procedure Open(Fd : Integer;
Mode : Integer := 0);
procedure Open(Id : Float;
Mode : Integer := 0);
Open(42); -- universal_integer-
>Integer & Float: AMBIGUOUS
```

In each case the SNA condition NAC-app fails. There are two coercion paths of equal rank from the argument's type to two different candidate types. None of the language-specific tie-breaking mechanisms resolves this conflict. The problem is with the ranking function rho itself, rather than the context Gamma or the expansion algorithm.

4.1.3 Differences between the Languages

Even though they all fail for the same fundamental reason, there are key differences. Java completely avoids Type-III ambiguities by not having default parameters (Gosling et al., 2023). Ada eliminates a great deal of Type-I and some Type-II ambiguities through its top-down type propagation rule (Ada-Auth, 2012). C# is the only language that has introduced a partial tiebreaker for default parameters into its specification (ECMA International, 2022). C++ is left with conversion-ranks only.

4.2 IMPLICATIONS FOR LANGUAGE DESIGN

4.2.1 Making rho a Total Order

The formalisation identifies the cause of all unresolvable ambiguities: ranking function rho does not define a strict order between two applicable coercion paths. A language design that ensures rho is a total ordering over all coercion paths (including user-defined conversions relative to each other) would ensure the NAC-app condition holds. Making the coercion graph acyclic and giving it a globally consistent topological order on its edges would guarantee SNA for all calls (Swamy et al., 2009).

In C# and C++ this could be achieved by requiring users to declare a priority for their user-defined implicit conversion operators. This would be similar to precedence statements in parser generators and would not limit the conversions that can be declared to only their interactions when they compete for the same argument position.

4.2.2 Default-Aware Candidate Ranking

The source of Type-III ambiguities is that default-parameter expansion produces synthetic candidates treated identically to programmer-written ones. A design that incorporates information about default parameters into the definition of dominance, favouring the declaration requiring fewest default

substitutions which would eliminate most Type-III ambiguities without impinging on overloading or default parameters. While C# already does a version of this (ECMA International, 2022), applying the rule more consistently would address the remaining discrepancy.

4.2.3 The Formal Meaning of Java's Decision

Java's choice to omit default parameters is an example of an important trade-off between expressiveness and a structurally ensured eradication of an entire ambiguity class. Scott (2016) notes that language design choices are inherently a trade-off between expressiveness, safety and predictability. Programming languages that want to provide default parameters while obeying SNA should employ the default-sensitive ranking improvement of Section 4.2.2.

V. CONCLUSIONS

This paper has offered a formal comparative analysis of the rules for resolving overloaded subprogram calls in C++, Java, C# and Ada, focusing particularly on the interaction between overloading, implicit type coercion and default parameters, to generate ambiguities that are not automatically resolved by the compilers.

Our analysis addressed the problem statement. The interaction of overloading, coercion and default parameters results in a pattern of ambiguity in all four languages — one where equally ranked coercion sequences compete for the same number of arguments, contra the SNA condition's NAC-app criterion. Java avoids Type-III ambiguities by disallowing default parameters, and Ada alleviates Type-I and some Type-II ambiguities by propagating the expected type top-down — but these still do not resolve the combination failure case of Section 4.1.2.

Two design suggestions were made: first, making the ranking function rho a total order on all coercion paths would ensure NAC-app for all calls; second, adding information about default parameters to the dominance predicate as a tie-breaking element would remove Type-III ambiguities without loss of expressiveness. Both extensions are consistent with the current definitions of the four languages and could be included in future revisions of their specifications.

The main contribution of this paper is to show that the SNA criterion and its formal terms (coercion relations, ranking functions, and type contexts) provide a formal language-independent way of determining and categorising resolution failures that may not be exposed by informal language specifications. Future research should mechanically prove the results using a proof assistant, and should address resolution issues in template and generic overloading.

REFERENCES

[1] Ada-Auth (2012) Annotated Ada Reference Manual: Language and Standard Libraries. Section 8.6. Ada Rapporteur Group / ISO/IEC 8652:2012. Available at: http://www.ada-auth.org/standards/aarm12_w_tc1/html/AA-8-6.html (Accessed: 1 April 2026).

[2] Allen, E., et al. (2008) 'On coercion for object-oriented programming languages', in Proc. OOPSLA '08. New York, NY: ACM, pp. 3–22.

[3] ECMA International (2022) ECMA-334: C# Language Specification. 6th edn. Geneva: ECMA International. Available at: https://www.ecma-international.org/publications-and-standards/standards/ecma-334/ (Accessed: 1 April 2026).

[4] ISO/IEC (2020) ISO/IEC 14882:2020 — Programming languages: C++ (6th ed.). International Organization for Standardization.

[5] Luo, Z. (2008) 'Coercions in a polymorphic type system.' Mathematical Structures in Computer Science, 18(4), pp. 729–751.

[6] Scott, M. L. (2016) Programming language pragmatics (4th ed.). Morgan Kaufmann. ISBN: 978-0124104099.

[7] Stroustrup, B. (2013) The C++ programming language (4th ed.). Addison-Wesley. ISBN: 978-0321563842.

[8] Swamy, N., et al. (2009) 'A Theory of Typed Coercions and Its Applications.' ACM SIGPLAN Notices, 44(9), pp. 329–340. https://doi.org/10.1145/1631687.1596598.