Value-Based Overload Resolution in Typed Languages

Abstract

This paper proposes a language feature extending traditional function overloading by allowing dispatch not only on parameter *types*, but also on *parameter values*. This form of polymorphism—"value polymorphism"—makes function selection sensitive to specified constant values and value ranges. We argue that this feature improves clarity, ensures completeness, and integrates naturally with static analysis, interface conformance, and unit test generation.

Motivation

In many real-world systems, especially financial and business rule applications, developers must select behavior based on both the *type* and *value* of parameters. Currently, this logic is expressed via switch statements, conditionals, or external rule engines. These approaches:

- Obscure the logical structure of the rule set
- Duplicate information across interfaces, implementations, and tests
- Introduce opportunities for inconsistency or error

By lifting value constraints into method signatures, the dispatch logic becomes *declarative*, *visible*, and *verifiable*.

Syntax Proposal

We propose extending function signatures with value constraints, expressed as constants, wildcards, or ranges:

```java
Decimal getRate(String accountType = "Platinum", int balance = 0..10000);
Decimal getRate(String accountType = "Platinum", int balance = 10001..20000);
Decimal getRate(String accountType = "Gold", int balance = *);
Decimal getRate(String accountType = "*", int balance = *);
```

- `"Platinum"` is a constant value constraint
- `*` is a wildcard (matches any value of the declared type)
- `0..10000` is a range constraint (inclusive)

Dispatch Semantics

Given a set of overloads, a method call is dispatched to the first implementation that matches all parameter constraints:

- **Type match** is required
- **Value match** requires:
- Exact match for constants
- Inclusion for range constraints
- Always true for wildcards

Formally, for method `$f_i$` with constraints `$c_1..c_n$` and runtime values `$v_1..v_n$`:

```
$$\text{Match}(f_i, v_1..v_n) \Leftrightarrow \forall i, \text{typeof}(v_i) = T_i \land v_i \text{ matches } c_i$$
```

Disjointness (Determinism)

Overloads must be disjoint:

```
$$\forall f_i \neq f_j, \neg\exists v \text{ such that } \text{Match}(f_i, v) \land \text{Match}(f_j, v)$$
```

Completeness (Optional)

Implementations may be required to cover the full input space:

```
$$\forall v \in T_1 \times ... \times T_n, \exists f_i \text{ such that } \text{Match}(f_i, v)$$
```

This enables interfaces to act as *decision tables* with total coverage.

Interface Semantics

Interfaces define a group of overloads forming a dispatch matrix. Implementations must:

- Match all declared overloads
- Provide no extraneous overloads
- Be statically verifiable as complete, disjoint, and type-correct

This tightens the contract between interface and implementation, eliminating ambiguity.

Testing and Verification

A powerful consequence of value-based overloads is that they can be **automatically transformed into unit tests**:

```java
@Test
void platinum_5000() {
   assertEquals(new BigDecimal("0.05"), impl.getRate("Platinum", 5000));
}
```

Since each overload defines a concrete behavioral region, it maps directly to testable examples. Test frameworks can:

- Generate tests from signatures
- Ensure implementation conforms to declared behavior
- Surface coverage gaps visually

Related Work

- **Pattern matching** (Scala, C#, F#) covers value-based logic but lacks interface integration

- **Rule engines** (Drools, OpenRules) provide declarative logic but require external tooling and runtime

- **Multi-method dispatch** (CLOS, Julia) resolves on runtime types, not values

- **Template specialization** (C++) supports value-based behavior only at compile time

Our approach preserves the rigor and clarity of Java-like interfaces while introducing declarative, verifiable value logic.

Implementation Strategies

We envision multiple possible implementation paths:

- Compiler transformation into `switch` or `if` statements
- Generation of dispatch tables at compile time
- Bytecode weaving or use of `invokedynamic`

Tooling Potential

- IDEs can render interfaces as decision tables
- Static analysis tools can detect ambiguous or unreachable overloads
- Test generators can instantiate test cases from signature constraints

Attribution

This concept was developed in 2025 by Steve Magner, as an extension of typed method dispatch toward declarative logic and test-integrated interfaces. For feedback or collaboration, contact via GitHub (username stevemagner).

License