Static Type Checking of Multi-Methods

Rakesh Agrawal Linda G. DeMichiel Bruce G. Lindsay

IBM Almaden Research Center
San Jose, California 95120

Abstract

Multi-methods allow method selection to be based on the types of any number of arguments. Languages that currently support multi-methods do not support static type checking. We show how multi-methods can be statically type checked and how information collected at the time of program compilation can be used to make the run-time dispatch of multi-methods more efficient. The results presented can provide the basis for introducing multi-methods in languages with static type checking and for designing new object-oriented paradigms based on multi-methods.

1 Introduction

Most programming languages support the notion of *typed* data. A data type consists of a representation and a set of operations that can be applied to instances of the type. In many object-oriented languages, types are organized in a hierarchy and a *subtype relation* is defined over them. The meaning of this subtype relation usually corresponds to that of *subtype polymorphism*, or *inclusion polymorphism*[4]: A is a subtype of B, A and B not necessarily distinct, exactly when every instance of A is also an instance of B. Operations on the instances of types are defined by *generic functions*, where a generic function corresponds to a set of *methods* and the methods define the type-specific behavior of the generic function. A method is defined for a set of arguments of particular types and can be executed for any objects that are instances of those types or their subtypes. The selection of the method to be executed depends on the types of the actual arguments with which the generic function is invoked. In the presence of subtype polymorphism, this method selection must, in general, occur at run time.

While methods in many object-oriented languages implement multiary operations, in most of these languages, methods are "selfish." That is, there is a single, distinguished argument whose type determines which method is executed when a generic function is called. The remaining arguments are ancillary: they provide values for the method that is selected, but they play no role in method selection itself. Thus, for example, while it is possible in C++ [7] to write a virtual function of the form `float area(shape)`[1] that is dynamically dispatched based on the actual type of `shape` supplied with the function invocation, one cannot write a virtual function of the form `void draw(window, shape)` that is dynamically dispatched based on actual types of both `window` and `shape`.

Multi-methods [3] are a generalization of selfish methods: they allow method selection to be based on the types of all arguments, not just a single argument. Selfish methods can be simulated by multi-methods by associating all but one formal method argument with a most general type. However, in order to simulate multi-methods in a system that does not provide them, the user must resort to awk-

[1] The exact syntax is different in C++; we have taken liberty with the syntax to clarify the point.

ward techniques such as currying [5, 8].

The integration of multi-methods and statically-typed languages is not well understood. Languages that support multi-methods (e.g., the Common Lisp Object System (CLOS) [2] and its ancestor CommonLoops [3]) do not perform static type checking.

In this paper, we present mechanisms for the static type checking of multi-methods. In addition, we show how these mechanisms allow multi-methods to be more efficiently dispatched at run time. The results presented are intended to provide the basis for introducing multi-methods in statically-typed languages and for designing new object-oriented paradigms based on multi-methods.

The work described here has been performed in the context of Polyglot, a multi-lingual object-oriented database type system that we are designing at the IBM Almaden Research Center. In database type systems, types are persistent and evolving. Type and method definitions come from many sources, such as type library vendors, application developers, and end users. This leads to a separation of processing into three phases:

1. *Type definition.* This phase is akin to schema definition in database parlance. The definitions of types and methods are processed in this phase.
2. *Compilation and type checking.* During program compilation, every generic function invocation is type checked to insure that it cannot result in a run-time type error.
3. *Execution and run-time dispatch.* At run time, appropriate methods are selected for execution on the basis of the actual arguments of the function invocations.

1.1 Organization of the Paper

The organization of this paper is as follows. We formally define the problem of the static type checking of multi-methods in Section 2. The main source of difficulty is the following: Because of subtype polymorphism, there can be more than one method that is applicable for a generic function invocation, and it is not always possible to determine at compile time which method will be executed at run time.[2] Given a generic function invocation, and a set of methods that are applicable for that invocation, a precedence relationship is needed to select the method that in some sense most closely "matches" the invocation. Section 3 discusses various rules for establishing such method precedence. Section 4 describes our methodology for the static type checking of multi-methods. This consists of two steps. First, at the time of method definition, it is insured that the methods of a generic function are mutually consistent. Sufficient conditions for insuring such consistency are given in Section 5. Next, at the time of program compilation, every static function invocation is type checked to determine its validity. This step is described in Section 6. Section 7 discusses the run-time dispatch of multi-methods. We discuss related work in Section 8. Section 9 summarizes the results of our work.

1.2 Notation

In the discussion that follows, we represent the subtype relation by $\preceq$. If $A \preceq B \wedge A \neq B$, we say that A is a *proper subtype* of B and represent this relation by $\prec$. Since the proper subtype relation is a partial order, we can view such a system of types as a directed acyclic graph. There is a path from A to B if and only if A is a subtype of B. If $A \preceq B$, we also say that B is a *supertype* of A.

We will use upper case letters to denote types and the corresponding lower case letters to denote their instances. Thus, we will write a to denote an instance of type A.

We will use upper case letters to denote generic functions. We will denote a particular method m_k of an n-ary generic function M as $m_k(T_k^1, T_k^2, \ldots, T_k^n) \rightarrow R_k$, where T_k^i is the type of the i^{th} formal argument of method m_k, and where R_k is the type of the result. We will sometimes not

[2] This problem also exists for languages supporting only selfish methods. The solutions we present apply to such languages as well.

write the result type of such a method. The call to the generic function will be denoted without a subscript on m.

Finally, in our figures, we will draw an arrow from subtype to supertype to denote the subtype relationship.

2 Formal Statement of the Problem

Static type checking in the absence of polymorphism requires that we insure at compile time that every operation receives the proper number of arguments and that these arguments are instances of the proper data types. To allow for subtype polymorphism, this constraint is relaxed in the following way: We assume that it is admissible for a variable or a formal argument of a function to be bound at run time to a value that is an instance of its static type or an instance of any subtype of its static type. However, because instances of subtypes may be substituted for instances of supertypes at run time, the method that is selected for execution at run time for a given generic function invocation may be different from the method that would have been selected were the arguments of the same types as those of the static call. Therefore, for each generic function invocation, we need to be able to determine that there exists some method whose formal argument types are supertypes of the types of the static argument types, that the run-time type of each actual argument will be a subtype of the type of the corresponding static argument, and that the run-time type of the result will be consistent with the context in which it occurs.

In the following, we consider the type checking of expressions of the form

$$r : R \leftarrow f(g(\ldots h(m(\ldots), \ldots), \ldots), \ldots).$$

In the case in which the result of the generic function invocation is assigned to a variable, the result type of the function invocation must be a subtype of the type of that variable. In the case where the result of the generic function invocation is passed as an argument to an enclosing function invocation, the result type of the nested function invocation must be acceptable as the type of the corresponding argument of the enclosing function invocation.

Formally, given a generic function M, where M consists of the set of methods $\{m_1, \ldots, m_m\}$,

$$\begin{gathered} m_1(T_1^1, T_1^2, \ldots, T_1^n) \rightarrow R_1 \\ \vdots \\ m_m(T_m^1, T_m^2, \ldots, T_m^n) \rightarrow R_m \end{gathered}$$

and given a static generic function invocation

$$m(T^1, \ldots, T^n)$$

where $T^1, \ldots, T^n$ are the static types of the arguments and are known, the problem is to insure at compile time that the following two conditions hold:

1. There is at least one method of M that can be selected for execution at run time for arguments of types $T^1, \ldots, T^n$.
2. If a method $m_k(T_k^1, T_k^2, \ldots, T_k^n) \rightarrow R_k$ can be selected for execution at run time due to the occurrence of instances of subtypes of any of the static argument types, then:

   a. If the result of the function invocation $m(T^1, \ldots, T^n)$ is assigned to a variable

   $$r : R \leftarrow m(T^1, \ldots, T^n)$$

   then the result type R_k is a subtype of the type R of the receiver r.

   b. If the result of the function invocation $m(T^1, \ldots, T^n)$ is passed as an argument to an enclosing function invocation

   $$\tilde{m}(\tilde{T}^1, \ldots, m(T^1, \ldots, T^n), \ldots, \tilde{T}^p)$$

   then it can be verified that conditions (1) and (2) hold for the static invocation $\tilde{m}(\tilde{T}^1, \ldots, R_k, \ldots, \tilde{T}^p)$.

Condition (1) guarantees that the situation can never arise at run time in which there is no method that is applicable for the actual arguments of the generic function. Condition (2) guarantees that no matter which method is executed, its result will be consistent with the context in which it occurs. If we are able to guarantee that these conditions hold for all generic function invocations, then it is possible to guarantee at compile time that no invocation of a generic function will result in a type error at run time.

To simplify our presentation in the discussion that follows, we assume that all methods of a generic function have the same number of arguments, referred to as the *arity* of the function. This requirement is not necessary provided that each method has a fixed number of arguments, since the methods can be partitioned into equivalence classes by number of arguments. We also assume that no two methods of a generic function are permitted to have identical formal argument types, otherwise it would be impossible to select between them solely on the basis of the argument types of the function invocation. Our discussion is neutral with regard to the issue of whether generic functions are themselves objects or whether a generic function consists solely of a set of methods.

3 Method Applicability and Method Specificity

Because of subtype polymorphism, there can be more than one method that is applicable for a generic function invocation, and it is not possible to determine at compile time exactly which method will be executed at run time.

3.1 Method Applicability and Confusability

Given a generic function invocation, $m(T^1, \ldots, T^n)$, we say that a method $m_k(T_k^1, \ldots, T_k^n)$ is *applicable* for that invocation if and only if $\forall i,\ 1 \leq i \leq n,\ T^i \preceq T_k^i$.

If two methods are both applicable for some function invocation, we say that they are *confusable*. Formally, methods $m_1(T_1^1, \ldots, T_1^n)$ and $m_2(T_2^1, \ldots, T_2^n)$ are *confusable* if $\forall i,\ 1 \leq i \leq n$, there exists a type T^i, such that $T^i \preceq T_1^i \wedge T^i \preceq T_2^i$; otherwise they are *non-confusable*.

Note that confusability is not transitive. Consider the following type hierarchy and the meth-

$$
\begin{tikzcd}
A & & B & & C \\
& D \arrow[ul] \arrow[ur] & & E \arrow[ul] \arrow[ur] &
\end{tikzcd}
$$

ods $m_1(A)$, $m_2(B)$, and $m_3(C)$. Methods $m_1(A)$ and $m_2(B)$ are confusable, and methods $m_2(B)$ and $m_3(C)$ are confusable, but methods $m_1(A)$ and $m_3(C)$ are not confusable.

3.2 Method Specificity

In general, more than one method can be applicable for a generic function invocation. Given a set of methods that are applicable for a generic function invocation, a precedence relationship is needed to select the method that in some sense most closely *matches* the invocation. If one method has precedence over another for a given invocation, we say that it is *more specific*.

Given a set of confusable methods, we can identify the following representative mechanisms for determining method specificity, drawing upon the design choices in current object-oriented languages: i) arbitrary global precedence, ii) argument subtype precedence, iii) argument order precedence, iv) global type precedence, v) inheritance order precedence, and vi) arbitrary local precedence.

The first four of these mechanisms are global in that they establish a global precedence relationship over a set of confusable methods. The last two are local in that they allow precedence relationships to be different for instances of different types. These mechanisms are not all equally powerful. We discuss their relative advantages and disadvantages below. Table 1 summarizes their relative power in determining method specificity.

3.2.1 *Arbitrary Global Precedence*

The user is free to order confusable methods in any way the user pleases.

	single inheritance selfish methods	single inheritance multi-methods	multiple inheritance selfish methods	multiple inheritance multi-methods
arbitrary global prececence	$\surd$	$\surd$	$\surd$	$\surd$
argument subtype precedence	$\surd$			
argument order precedence	$\surd$	$\surd$		
global type precedence	$\surd$	$\surd$	$\surd$	$\surd$
inheritance order precedence	$\surd$	$\surd$	$\surd$	$\surd$
arbitrary local precedence	$\surd$	$\surd$	$\surd$	?

Table 1: Relative power of method precedence orderings

While simple, this approach is undesirable for several reasons. Aside from the burden it imposes on the user, it may permit orderings that violate tacit notions of method specificity, or orderings in which certain methods can never be executed. If the user is not careful, cycles may be created in the precedence relationship.

We are not aware of any language that allows a completely arbitrary global ordering of confusable methods.

3.2.2 Argument Subtype Precedence

The subtype relationships between the corresponding formal argument types of two confusable methods $m_i(T_i^1, T_i^2, \ldots, T_i^n)$ and $m_j(T_j^1, T_j^2, \ldots, T_j^n)$ are used to establish the precedence between those methods. If $\forall k$, $1 \leq k \leq n$, $T_i^k \preceq T_j^k$, and $\exists l$, $1 \leq l \leq n$, such that $T_i^l \prec T_j^l$, we say that m_i and m_j are in an *argument subtype relationship* and that m_i *precedes* m_j.

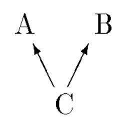
Consider, the types A and B, where $B \prec A$, and the confusable methods $m_1(A, A, B)$ and $m_2(A, B, B)$. Since all the formal argument types of m_2 are subtypes of the corresponding formal argument types of m_1 with the second argument type being additionally a proper subtype, m_2 is more specific than m_1.

We show in the appendix that argument subtype precedence is sufficient to establish precedence between confusable methods in a language that supports only single inheritance and selfish method selection.

As an example of such a language, we cite the versions of C++ [12] that predated the introduction of multiple inheritance to that language.

In a single inheritance language with multi-methods, however, argument subtype precedence is not sufficient to order confusable methods. Consider the types A and B, where $B \prec A$, and the confusable methods $m_1(A, B)$ and $m_2(B, A)$. The invocation $m(b, b)$ can be resolved neither to $m_1(A, B)$ nor to $m_2(B, A)$ using only argument subtype precedence.

Argument subtype precedence is likewise not sufficient to order confusable methods in a selfish method language that supports multiple inheritance. Consider the following type hierarchy and the methods $m_1(A)$ and $m_2(B)$. The invocation $m(c)$ can be resolved neither to $m_1(A)$ nor to $m_2(B)$ using only argument subtype precedence.



3.2.3 Argument Order Precedence

An *argument order* is a total order over the argument positions of the methods of a generic function. It may correspond to the left-to-right ordering of arguments or it may be any permutation of that order. The method $m_i(T_i^1, \ldots, T_i^n)$ precedes the method $m_j(T_j^1, \ldots, T_j^n)$ according to a given argument order if there is some argument position k, $1 \leq k \leq n$, such that $\forall l$, $1 \leq l < k$, $T_i^l = T_j^l$, and $T_i^k \prec T_j^k$.

Consider the types A and B, where $B \prec A$, the confusable methods $m_1(A,B,A,B)$ and $m_2(A,B,B,A)$, and left-to-right argument order. The formal argument types of the two methods differ first in the third position, where the formal argument type of m_2 is a proper subtype of the formal argument type of m_1. Therefore, m_2 is more specific than m_1 according to argument order precedence.

We show in the appendix that argument order precedence is sufficient to establish precedence between confusable methods in a language that supports multi-methods but admits only single inheritance.

In the case of multiple inheritance, confusable methods can have formal argument types that are not in any subtype relationship. Argument order precedence is not sufficient to determine method precedence in this case. Consider again the type hierarchy shown in the figure above and the methods $m_1(C,A)$, $m_2(C,B)$. The invocation $m(c,c)$ can be resolved neither to $m_1(C,A)$ nor to $m_2(C,B)$.

3.2.4 Global Type Precedence

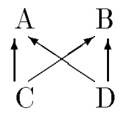
A *global type order* is a partial order α, such that i) if $A \prec B$, then $A \,\alpha\, B$, and ii) if there is a type C such that $C \,\alpha\, D$ and $C \,\alpha\, E$, then either $D \,\alpha\, E$ or $E \,\alpha\, D$.

Given any two confusable methods $m_i(T_i^1, T_i^2, \ldots, T_i^n)$ and $m_j(T_j^1, T_j^2, \ldots, T_j^n)$, consider their formal argument types in a prespecified order (such as left-to-right) and find the first argument position in which the formal argument types of m_i and m_j differ, say k. Method m_i is more specific than method m_j according to *global type precedence* if and only if $T_i^k \,\alpha\, T_j^k$ in the global type ordering.

Consider, again the type hierarchy shown in the figure above and the methods $m_1(C,A)$ and $m_2(C,B)$. If $A \,\alpha\, B$, then m_1 is more specific than m_2.

We show in the appendix that global type precedence is sufficient to order confusable methods in a multiple inheritance language with multi-methods.

Note that the global type ordering need not be a total order. For example, consider the following type hierarchy. Types C and D need not be ordered with respect to each other.



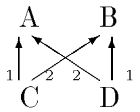
3.2.5 Inheritance Order Precedence

In a global type ordering, the precedence between two types is required to be the same for all common subtypes of those types. In a local type ordering, on the other hand, the precedence between the supertypes of a type can be freely chosen as long as the precedence relationships defined by their supertypes are not violated. Thus, the precedence between two types need not be the same for all of their common subtypes. The precedence between the supertypes of a type T may be explicitly specified, or it may be inferred from the lexical order in which they occur in the definition of type T.

Unlike the previous mechanisms, where method specificity depends on the types of the formal arguments of the method definitions, *inheritance order precedence* requires method specificity to be determined on the basis of the types of the arguments of the generic function invocation. Given two methods $m_i(T_i^1, T_i^2, \ldots, T_i^n)$ and $m_j(T_j^1, T_j^2, \ldots, T_j^n)$ that are applicable for a generic function invocation $m(T^1, T^2, \ldots, T^n)$, consider their formal arguments in a prespecified order (such as left-to-right) and find the first argument position in which the formal argument types of m_i and m_j differ, say k. If $T_i^k \propto T_j^k$ in the local type ordering of T^k, then m_i is more specific than m_j, and vice versa.

Formally, a *local type ordering for a type* C is a total order $\propto_C$ over C and its supertypes such that if $C \prec A$ and $C \prec B$, then $C \propto_C A$, $C \propto_C B$, and either $A \propto_C B$ or $B \propto_C A$. Furthermore, if $C \prec B$, and if $D \propto_B E$ in the local type ordering for type B, then $D \propto_C E$ in the local type ordering for the type C, and this rule is recursively applied.

For example, consider the following type hierarchy and the methods $m_1(A)$ and $m_2(B)$. We



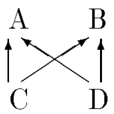
have that $A \propto_C B$ in the local type ordering for type C and that $B \propto_D A$ in the local type ordering for type D. Method $m_1(A)$ is more specific than method $m_2(B)$ for the invocation $m(c)$, while method $m_2(B)$ is more specific than method $m_1(A)$ for the invocation $m(d)$.

We show in the appendix that inheritance order precedence is sufficient to determine method specificity in a multiple inheritance language with multi-methods.

CLOS is an example of a language that uses inheritance order precedence to determine method specificity.

3.2.6 Arbitrary Local Precedence

C++ allows method precedence to be explicitly specified by the user as part of type definition. We call this mechanism for resolving method specificity *arbitrary local precedence*. For example, consider again the following type hierarchy and the methods



$m_1(A)$ and $m_2(B)$. In C++, it is possible to specify in the definition of type C that method $m_1(A)$ is more specific than method $m_2(B)$, and in the definition of type D that method $m_2(B)$ is more specific than method $m_1(A)$. The invocation $m(c)$ causes $m_1(A)$ to be dispatched, while the invocation $m(d)$ causes $m_2(B)$ to be dispatched.

This approach is feasible in C++ because C++ methods are selfish methods. However, arbitrary local precedence does not appear to generalize cleanly to multi-methods.

4 Static Type Checking of Multi-Methods

Having discussed mechanisms for establishing a method precedence order, we can now state the central result.

Theorem 1 *Multi-methods can be statically type checked as follows:*

1. *Insure that all method definitions are mutually consistent.*

   Two methods $m_i(T_i^1, T_i^2, \ldots, T_i^n) \rightarrow R_i$ and $m_j(T_j^1, T_j^2, \ldots, T_j^n) \rightarrow R_j$ of a generic function M are *mutually consistent* if whenever they are both applicable for arguments of types $T^1, \ldots, T^n$ and m_i is more specific than m_j, then $R_i \preceq R_j$. A generic function is consistent if all of its methods are mutually consistent.
2. *For each static generic function invocation $m(T^1, \ldots, T^n)$, apply the following procedure:*

   a. *Determine that there is at least one method that is applicable for the invocation.*

   b. *Find the definition of the most specific applicable method m_k for the argument types $T^1, \ldots, T^n$.*

   c. *Verify for the method $m_k(T_1^k, \ldots, T_n^k) \rightarrow R_k$ selected in step (2b) that:*

      i. *If the result of the function invocation $m(T^1, \ldots, T^n)$ is assigned to a variable*

      $$r : R \leftarrow m(T^1, \ldots, T^n)$$

      then $R_k \preceq R$.

      ii. *If the result of the function invocation $m(T^1, \ldots, T^n)$ is passed as an argument to an enclosing function invocation*

      $$\tilde{m}(\tilde{T}^1, \ldots, m(T^1, \ldots, T^n), \ldots, \tilde{T}^p)$$

      then step (2) is applied to the static invocation

      $$\tilde{m}(\tilde{T}^1, \ldots, R_k, \ldots, \tilde{T}^p).$$

Step (1) is performed once at method definition time. Step (2) is performed at compile time for each generic function invocation.

Proof: If step (2a) yields a method m_i, we are guaranteed that there will be at least one method of M that can be selected for execution at run time. Let m_k be the most specific applicable method for the static argument types selected in step (2b). If method m_r, different from m_k, is selected at run time, then m_r will be more specific than m_k for the actual argument types, and the definition of method consistency implies that $R_r \preceq R_k$. Since step (2ci) guarantees that $R_k \preceq R$, it follows from the transitivity of the subtype relationship that $R_r \preceq R$ and there cannot be a run-time type error in the case of assignment. The correctness of step (2cii) can be shown by induction on the structure of the expression. ∎

We now discuss mechanisms for achieving steps (1) and (2).

5 Conditions for a Consistent System

Before presenting conditions for insuring that a generic function is consistent, we need the definitions of confusable set, method graph, method precedence graph, result graph, and graph conformance.

A *confusable set* of methods is a maximal set C such that methods m_i and m_j are in C if there are $k \geq 0$ methods $m_1, m_2, \ldots, m_k$ such that m_i is confusable with m_1, m_1 is confusable with $m_2, \ldots,$ and m_k is confusable with m_j. If a method is not confusable with any other method, it forms a singleton confusable set. The confusable sets over a set of methods M thus disjointly partition M.

A *method graph* for a confusable set is an undirected graph such that

1. For every method m_i in the confusable set, there is a node labeled by that method.
2. There is an edge between the nodes labeled m_i and m_j if and only if m_i and m_j are confusable.

A *method precedence graph* for a confusable set is a directed graph such that

1. For every method m_i in the confusable set, there is a node labeled by that method.
2. There is an arc from node m_i to m_j if and only if m_i and m_j are confusable and m_i is more specific than m_j.

A method precedence graph induces a graph over the result types of the corresponding methods. A *result graph* for a confusable set is a directed graph such that

1. For every result type R_i that is a result type of some method m_i in the confusable set, there is a node labeled by that result type.
2. There is an arc from node R_i to R_j if and only if there is an arc from node m_i to m_j in the method precedence graph.

A directed graph G_1 *conforms* to a directed graph G_2 if for every arc from node i to node j in G_1, there is a path in G_2 from i to j.

5.1 Sufficient Conditions

Theorem 2 *A generic function is consistent if the following two conditions both hold for every confusable set:*

1. *For every edge between two nodes in the method graph, there is at least one arc between those same nodes in the method precedence graph.*
2. *The result graph conforms to the type graph.*

Proof: First, from the maximality of confusable sets, no two methods of a generic function belonging to different confusable sets can be confusable, and hence by definition all such methods are mutually consistent.

For two methods m_i and m_j belonging to the same confusable set, condition (1) guarantees that if they are both applicable for some argument types, then the precedence between them has been established. Condition (2) guarantees that if m_i is more specific than m_j, then the result type of m_i is a subtype of the result type of m_j. ∎

Corollary 1 *A generic function is consistent if every confusable set of this function is a singleton.*

Corollary 2 *All methods in a confusable set of a generic function are mutually consistent if every method has the same result type.*

Theorem 2 implies that if the method precedence graph has a cycle, the methods involved in the cycle must have the same result type for condition (2) to hold. A cycle can arise in the method precedence graph if a local mechanism such as inheritance order precedence is used to resolve method specificity, as is the case in CLOS.

In a system that requires the condition stated in Corollary 1 to hold, many methods can be associated with a given generic function, but at most one method can ever be applicable for any function invocation. Such a system has no need for recourse to run-time discrimination for method selection, since if there is an applicable method for a given static function invocation, it is always unique. It is necessary only to verify at compile time that an applicable method exists and that its result type is consistent with the expression context of the function invocation. Ada is an example of a language that supports *function overloading* in this sense, but not run-time discrimination.

5.2 Verification of Method Consistency

We now present an algorithm for the construction of the method precedence graph of a generic function and the verification of the consistency of its method definitions. The algorithm for this step, *ProcessMethods*, uses three subroutines:

1. *Confusable*: This subroutine determines whether two given methods are confusable.
2. *ComputePrecedence*: This subroutine determines which of two given methods is the more specific.
3. *CheckConsistency*: This subroutine determines whether two given methods are consistent and creates the method precedence graph.

```
procedure ProcessMethods(generic-function)
  let m be the number of methods of generic-function
  for i in 1 to m do
    for j in i+1 to m do
      /* check if m_i and m_j are confusable */
      if Confusable(m_i, m_j) then
        /* find the relative ordering of m_i and m_j */
        order := ComputePrecedence(m_i, m_j)
        /* check if m_i and m_j are consistent */
        if CheckConsistency(m_i, m_j, order) =
          Inconsistent then
            Output("m_i, m_j are inconsistent")
    od
  od

function Confusable(m_i, m_j)
      returns {True, False}
  let the argument types of m_i be T_i^1, ..., T_i^n
  let the argument types of m_j be T_j^1, ..., T_j^n
  for k in 1 to n do
    if T_i^k and T_j^k have no common subtype then
      return False /* m_i and m_j not confusable */
  od
  return True /* m_i and m_j are confusable */

function ComputePrecedence(m_i, m_j)
      returns {◁, ▷, ◇}
  let n be the arity of m_i and m_j
  let the argument types of m_i be T_i^1, ..., T_i^n
  let the argument types of m_j be T_j^1, ..., T_j^n
  for k in 1 to n in the prespecified argument order do
    if T_i^k is not the same as T_j^k then
      /* this position determines the ordering */
      if T_i^k < T_j^k then
        return ◁
      else if T_j^k < T_i^k then
        return ▷
      else /* local precedence ordering case */
        return ◇
  od

function CheckConsistency(m_i, m_j, order)
      returns {Consistent, Inconsistent}
  if order = ◁ then /* m_i more specific than m_j */
    if R_i ≰ R_j then
      return Inconsistent
    else create an arc from m_i to m_j
      in the method precedence graph
  else if order = ▷ then /* m_j more specific than m_i */
    if R_j ≰ R_i then
      return Inconsistent
    else create an arc from m_j to m_i
  else /* order = ◇ */
    /* cannot order m_j and m_i */
    if R_j ≠ R_i then
      return Inconsistent
    else create an arc from m_i to m_j
      and another from m_j to m_i
  return Consistent
```

5.2.1 Remarks

1. Functions *Confusable* and *ComputePrecedence* both examine the formal argument types of the two methods under consideration in some specified order. In an actual implementation, the two subroutines may be merged into one for efficiency. We have not done so in the description above for ease of exposition.

2. The function *ComputePrecedence* is dependent on the mechanism used by the language for determining method specificity. If arbitrary global precedence or arbitrary local precedence is used, the precedence between every two confusable methods is specified by the user. We do not discuss these cases further. In all other cases, the relationships between the types of corresponding formal arguments of confusable methods is used to determine precedence. We define a relationship $<$ on distinct types in the system. For argument subtype precedence and argument order precedence, $<$ is the same as the proper subtype relationship $\prec$. For global type precedence and inheritance order precedence, we define an augmented type graph and define $<$ as any topological sort on this augmented type graph. For global type precedence, the augmented type graph consists of the type graph representing the proper subtype relationship augmented with arcs representing the global type order α. For inheritance order precedence, the augmented type graph consists of the type graph representing the proper subtype relationship augmented with arcs representing all local type orderings α. In the latter case, the augmented type graph may contain cycles, and types in each strongly connected component are coalesced into a virtual node before the topological sort.

3. In *CheckConsistency*, instead of forming a result graph and checking that it conforms to the type graph, we check that $R_i \preceq R_j$ whenever an arc is to be created from m_i to m_j in the method precedence graph. The following lemma guarantees that this procedure is correct:

   Lemma 1 *The result graph conforms to the type graph if for every arc from R_i to R_j in the result graph, $R_i \preceq R_j$.*

   Proof: Follows from the transitivity of the subtype relationship. ∎

4. Determining whether two methods are confusable requires testing whether corresponding formal argument types have a common subtype. Testing whether the result graph conforms to the type graph also depends on determining whether two types are in a subtype relationship. In order to make subtype relationship tests fast, we maintain a compressed transitive closure of the subtype relationship using the technique described in [1]. An index and a set of ranges are associated with each type. If the index of one type falls into a range of another type, then the first type is a subtype of the second. Furthermore, if any of the ranges associated with two types overlap, then the two types have common subtypes in the overlapping range. Using this technique, we can test subtype relationships in constant time if the type hierarchy is a directed tree (single inheritance). If the type hierarchy is a directed acyclic graph (multiple inheritance), experimental results show that subtype relationships can be tested in essentially constant time[1], and, in the worst case (bipartite graph), in $O(log(n))$ time, where n is the number of types in the system.

5.2.2 Complexity of the Algorithm

ProcessMethods calls functions *Confusable* and *ComputePrecedence* and *CheckConsistency* each $O(m(m+1)/2)$ times, where m is the number of methods of the generic function. Functions *Confusable* and *ComputePrecedence* each require $O(n)$ argument type comparisons, where n is the arity of the generic function. Function *CheckConsistency* requires one subtype test.

5.3 Construction of the Confusable Sets from the Method Precedence Graph

The confusable sets for the generic function correspond to the connected components of the method precedence graph. They can be obtained in time linear in the number of nodes and edges of the method precedence graph by using Tarjan's algorithm[13]. Tarjan's algorithm yields, at the same time, a topological sort of the methods in each confusable set such that if method m_i is more specific than m_j, then method m_i precedes m_j in the sort order. If the method precedence graph contains cycles, Tarjan's algorithm condenses non-trivial strongly connected components (methods involved in cycles) into virtual nodes, which we call *blobs*, and yields a topological sort of the resulting acyclic graph.

6 Checking the Validity of Generic Function Invocations

In the previous section, we discussed the validation of method consistency that is performed at type definition time. We now turn to a discussion of the type checking of generic function invocations that is done at compile time. Recall from Theorem 1 that this requires finding the most specific applicable method for the static argument types and verifying that the result type of this method is consistent with its context.

Finding the most specific applicable method involves two steps:

1. Finding the confusable set in which such a method is contained.
2. Finding the most specific applicable method within that confusable set.

6.1 Finding the Confusable Set

We say that a generic function invocation $m(T^1, \ldots, T^n)$ is *covered by* a confusable set S if there exists a method $m_s \in S$ such that m_s is applicable for m.

Lemma 2 *A generic function invocation $m(T^1, \ldots, T^n)$ is covered by at most one confusable set.*

Proof (by contradiction): Assume that m is covered by two distinct confusable sets, say S_1 and S_2. This implies that there exists a method $m_{s_1} \in S_1$ and a method $m_{s_2} \in S_2$ such that both m_{s_1} and m_{s_2} are applicable for m. This in turn implies that $\forall i,\ 1 \leq i \leq n,\ T^i \preceq T^i_{m_{s_1}} \wedge T^i \preceq T^i_{m_{s_2}}$. This, however, implies that m_{s_1} and m_{s_2} are confusable, which contradicts the assumption that S_1 and S_2 are distinct confusable sets. ∎

Before presenting an algorithm for determining the confusable set which covers a generic function invocation, we need a definition and a lemma.

Given a confusable set S for a generic function M of arity n, and an argument position i, $1 \leq i \leq n$, we say that type $\hat{T}$ is an element of $tops(S, i)$ for M if and only if there exists a method $m_j(T_j^1, \ldots, T_j^n) \in S$ such that $\hat{T} = T_j^i$ and there is no method $m_k(T_k^1, \ldots, T_k^n) \in S$ such that $\hat{T} \prec T_k^i$.

Lemma 3 *If a generic function invocation $m(T^1, \ldots, T^n)$ is covered by a confusable set S, then $\forall i,\ 1 \leq i \leq n,\ \exists \hat{T}^i | \hat{T}^i \in tops(S, i) \wedge T^i \preceq \hat{T}^i$.*

Proof: If a generic function invocation $m(T^1, \ldots, T^n)$ is covered by a confusable set S, then by definition there exists a method $m(\tilde{T}^1, \ldots, \tilde{T}^n)$ in S such that $\forall i,\ 1 \leq i \leq n$, $T^i \preceq \tilde{T}^i$. By definition of *tops*, $\forall i,\ 1 \leq i \leq n$, $\exists \hat{T}^i | \hat{T}^i \in tops(S, i) \wedge \tilde{T}^i \preceq \hat{T}^i$. The lemma then follows from the transitivity of the subtype relationship. ∎

Note that this lemma provides a necessary, but not a sufficient condition for a generic function invocation be covered by a confusable set. Consider the following type hierarchy and the methods

$$\begin{array}{cc} A & B \\ \uparrow & \uparrow \\ C & D \end{array}$$

$m_1(A, D)$ and $m_2(C, B)$. Now consider the function invocation $m(a, b)$. Clearly there is no applicable method for this invocation, but the condition stated with the lemma is satisfied.

Similarly, it is possible for the *tops* sets of more than one confusable set to satisfy the lemma condition for a given invocation. Consider, for example, the following type hierarchy and the methods $m_1(A_1, A_2)$, $m_2(A_2, A_3)$,

$$\begin{tikzcd}
A_1 & & A_2 & & A_3 & & A_4 & & A_5 \\
& B_1 \arrow[ul] \arrow[ur] & & B_2 \arrow[ul] \arrow[ur] & & B_3 \arrow[ul] \arrow[ur] & & B_4 \arrow[ul] \arrow[ur] &
\end{tikzcd}$$

$m_3(A_3, A_4)$, and $m_4(A_1, A_5)$. There are two confusable sets: $CS_1 = \{m_1, m_2, m_3\}$ and $CS_2 = \{m_4\}$. Furthermore, $tops(CS_1, 1) = \{A_1, A_2, A_3\}$, $tops(CS_1, 2) = \{A_2, A_3, A_4\}$, $tops(CS_2, 1) = \{A_1\}$, and $tops(CS_2, 2) = \{A_5\}$. For the function invocation $m(b_1, b_4)$, the tops test is satisfied both by CS_1 and CS_2.

Thus, this lemma can be used to construct an algorithm that can be used to identify the confusable sets in which the most specific applicable method for a given invocation may *potentially* be found. Finding such a set does not guarantee that an applicable method actually exists for the invocation, nor does it guarantee that at most one such set will be found. However, if no such confusable set is found, then it is guaranteed that there is no applicable method.

6.1.1 Algorithm for Constructing Tops Sets

let c be the number of confusable sets
let n be the arity of the generic function
for i **in** 1 **to** c **do**
  let m_i be the number of methods in confusable set i
  for j **in** 1 **to** n **do**
    $tops(S_i, j) = \emptyset$
    for k **in** 1 **to** m_i **do**
      /* T_k^j is the type of the argument of method m_k at position j */
      if $\exists T_{S_i} \in tops(S_i, j)$ such that $T_k^j \preceq T_{S_i}$ **then**
        continue with next value of k
      else if $\exists T_{S_i} \in tops(S_i, j)$ such that $T_{S_i} \preceq T_k^j$ **then**
        /* T_k^j replaces all such $T_{S_i} \in tops(S_i, j)$ */
        $\forall T_{S_i} \in tops(S_i, j)$ such that $T_{S_i} \preceq T_k^j$,
          remove T_{S_i} from $tops(S_i, j)$
        add T_k^j to $tops(S_i, j)$
      else /* T_k^j is a new member of $tops(S_i, j)$ */
        add T_k^j to $tops(S_i, j)$
    od
  od
od

The *tops* sets are determined at method definition time. The complexity of this algorithm is linear in the total number of methods of the generic function. For each argument position of every method, the subtype relationship of the type of that formal argument is tested against the members of the corresponding *tops* set. These tests may require $2 \times |tops|$ comparisons.

6.1.2 Algorithm for Finding Confusable Sets for a Function Invocation

We now give the algorithm for finding the confusable sets in which the most specific applicable method for a given generic function invocation $m(T^1, \ldots, T^n)$ may potentially be found.

let c be the number of confusable sets
let n be the arity of the generic function
confusable-sets $= \emptyset$
for i **in** 1 **to** c **do**
  for j **in** 1 **to** n **do**
    if $\forall T \in tops(S_i, j)\ T^j \npreceq T$ **then**
      /* go to next confusable set */
      continue with next value of i
  od
  /* found a potential confusable set */
  add S_i to *confusable-sets*
od
return *confusable-sets*

The complexity of this algorithm is $O(c \times n \times |tops|)$, where $|tops|$ is the size of the largest *tops* set.

If more than one confusable set is identified, the algorithm for finding the most specific method is applied to each such set. Once an applicable method is found, the algorithm can terminate and need not examine other confusable sets because of Lemma 2.

6.2 Finding the Most Specific Applicable Method

We first consider the case where the method precedence graph for the confusable set is acyclic.

As we stated in Section 5.3, a by-product of the algorithm for the construction of confusable sets is a topological sort of the methods in each confusable set such that if method m_i is more specific than m_j then m_i precedes m_j in the sort order.

Given a confusable set and a generic function invocation, the following algorithm finds the most specific applicable method for that invocation, if such a method exists.

1. Examine the methods in the confusable set in the topological sort order.
2. If the method under consideration is applicable for the function invocation, stop; otherwise, continue to the next method.
3. If no applicable method is found, flag the invocation as invalid.

The complexity of this algorithm is $O(m_c \times n)$, where m_c is the number of methods in the confusable set and n is the function arity. The applicability test for a method is linear in number of arguments, and the test for each argument is a subtype relationship test.

We now consider the case where the method precedence graph for the confusable set contains cycles. Recall that this can occur when inheritance order precedence is used to determine method specificity.

As stated in Section 5.3, methods involved in cycles are grouped into virtual nodes, called *blobs*. We obtain a topological sort of the resulting acyclic graph, but do not order methods within blobs.

We again proceed in topological sort order. If the method under consideration is applicable and a non-blob method, it is the most specific applicable method for the given invocation. If the method under consideration is applicable and a blob method, it must be tested for specificity against all remaining applicable methods in the blob. If no applicable blob or non-blob method is found, the invocation is flagged as invalid. The time complexity of this procedure is still $O(m_c \times n)$.

Note that if the most specific applicable method is to be found for the purpose of validating the result type of the generic function invocation, and an applicable blob method has been identified, it is not necessary to examine additional methods in the blob, since as a result of Theorem 2, all methods in a blob are required to have the same result type.

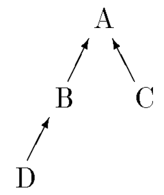
6.3 Validating the Result Type

Once the most specific applicable method for the static function invocation has been found, its result type is checked for consistency with the function invocation context as described in Section 4.

7 Run-time Dispatch

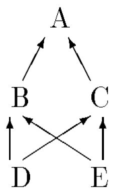
To improve the efficiency of the run-time dispatch of multi-methods, we save information obtained during the type checking of generic function invocations and associate it with these invocations for use at run time. Specifically, when examining the methods in the confusable set at compile time to find the most specific applicable method for a function invocation, we check for each method if it is confusable with the invocation. If it is confusable, it is saved; otherwise, it is discarded. Thus, a subset of the topological ordering is saved with the function invocation.

For example, consider the following type hierarchy and the confusable set consisting of



$m_1(A)$, $m_2(B)$, $m_3(C)$, and $m_4(D)$. A valid topological sort of these methods is $\{m_4(D), m_3(C), m_2(B), m_1(A)\}$. Given a static function invocation, $m(b)$, we save a subset of this topological sort, namely $\{m_4(D), m_2(B)\}$, with the invocation. We do not save $m_3(C)$ because it is not confusable with $m(b)$, and $m_1(A)$ is not saved because the search stops as soon as the first applicable method is found.

If the method precedence graph contains blobs, we may still need to examine all methods in these blobs at run time. For example, consider the following type hierarchy. Assume that $B \propto_D C$ in the



local type ordering for type D and that $C \propto_E B$ in the local type ordering for type E. Consider the confusable set consisting of $m_1(A)$, $m_2(B)$, and $m_3(C)$. Clearly, $m_2(B)$ and $m_3(C)$ are in a blob, and $m_1(A)$ is less specific than both of them. Given a function invocation, $m(a)$, we will need to save the blob consisting of $m_2(B)$ and $m_3(C)$ and examine it at run time because either $m_2(B)$ or $m_3(C)$ might be selected at run time, depending on whether the actual argument is of type D or E respectively.

8 Related Work

The use of multi-method function dispatch appears to have originated with CommonLoops [3]. The Common Lisp Object System (CLOS) [2] supports multi-method dispatch using inheritance order precedence. Type errors are detected at run time when no applicable method can be found for a function invocation.

Lécluse and Richard[10] characterize multi-method dispatch in terms of structural subtyping. Whenever two confusable methods are not ordered by argument subtype precedence, they require that additional methods be defined to insure that a most specific applicable method for any given set of arguments can always be determined by the use of argument subtype precedence alone. They do not discuss algorithms for static type checking or for method dispatch.

Mugridge, et al. [11] discuss multi-method dispatch with static type checking. They describe a method specificity rule that only partially orders the methods of a generic function. They define the *cover* of a method $m(T^1, \ldots, T^n)$ as the cross-product of all possible argument types: $\{< S^1, S^2, \ldots, S^n > \mid \forall i, S^i \preceq T^i\}$. Method applicability is defined in terms of the non-empty intersection of the covers of the function call and the method definitions. Given a call, applicable methods are found by intersecting corresponding covers. Since the method precedence rule does not totally order methods, if two applicable methods are found that do not have an order defined between them, such a call is declared ambiguous.

Finally, Dussud [6] and Kiczales and Rodriguez [9] describe techniques for efficiently dispatching multi-methods in CLOS. These techniques are based on hashing on the argument types. They select a method from the entire set of methods of the generic function. Because compile-time type checking is absent, the smaller set of potentially applicable methods cannot be used to reduce the run-time search space for method dispatch.

9 Conclusions

Until now, algorithms for the static type-checking of multi-methods with multiple inheritance were unknown.

We have demonstrated the feasibility of static type checking for multi-methods and have shown how the concept of *method confusability* can be used to classify the methods of a generic function.

This classification supports efficient identification of potentially applicable methods in large-scale type systems. The compile-time technique of identifying potentially applicable methods can be used to reduce the run-time search space for method dispatch.

In the course of our development of mechanisms for multi-method type checking, we have identified some language design choices for method specificity and have discussed their impact on the type checking and dispatch of both selfish methods and multi-methods. In particular, the distinctions between global and local method precedence rules and single and multiple inheritance have been shown to significantly affect the complexity of type-checking and method dispatch.

The results presented in this paper can provide the basis for adding multi-methods to existing languages. They can also be used when making choices in designing new object-oriented paradigms.

10 Acknowledgments

We would like to thank Alex Aiken, Laura Haas, Guy Lohman, and Jim Stamos for their helpful comments and suggestions.

11 Appendix

Lemma 4 *In a single inheritance system, if there is a type T such that $T \preceq T_1$ and $T \preceq T_2$, then either $T_1 \preceq T_2$ or $T_2 \preceq T_1$.*

Proof: In a single inheritance system, every type has at most one immediate supertype. ∎

Theorem 3 In a system that supports only single inheritance and selfish method selection, argument subtype precedence is sufficient to establish precedence between confusable methods.

Proof: Consider two confusable methods $m_1(T_1^1, T_1^2, \ldots, T_1^n)$ and $m_2(T_2^1, T_2^2, \ldots, T_2^n)$. Assume without loss of generality that method selection is based on the first argument. Since selfish method selection is used, it must be the case that $T_1^1 \neq T_2^1$. By the definition of confusability, there must be a common subtype of T_1^1 and T_2^1. It follows from Lemma A1 that either $T_1^1 \prec T_2^1$ or $T_2^1 \prec T_1^1$. ∎

Theorem 4 Argument order precedence is sufficient to establish precedence between confusable methods in a system that supports multi-methods but admits only single inheritance.

Proof: If methods $m_1(T_1^1, T_1^2, \ldots, T_1^n)$ and $m_2(T_2^1, T_2^2, \ldots, T_2^n)$ are confusable, then for any given T_1^i, T_2^i there must be a common subtype of T_1^i and T_2^i. In a single inheritance system, it follows from Lemma 4 that either $T_1^i \preceq T_2^i$ or $T_2^i \preceq T_1^i$. Consider the formal argument types of m_1 and m_2 in the specified argument order, and find the first argument position in which the argument types of m_1 and m_2 differ, say k. If $T_1^k \prec T_2^k$, then m_1 is more specific than m_2, and vice versa. ∎

Theorem 5 Global type ordering is sufficient to order confusable methods in a multiple inheritance system with multi-methods.

Proof: Consider the corresponding formal argument types of two confusable methods $m_1(T_1^1, T_1^2, \ldots, T_1^n)$ and $m_2(T_2^1, T_2^2, \ldots, T_2^n)$ in the specified argument order, and find the first position in which these types differ, say k. If $T_1^k \, \alpha \, T_2^k$, then m_1 is more specific than m_2, and vice versa. ∎

Theorem 6 Inheritance order precedence is sufficient to determine method specificity in a multiple inheritance system with multi-methods.

Proof: The proof is similar to that of Theorem 5 except that the local type ordering associated with the type of k^{th} argument of the function invocation is used to determine method specificity. ∎

References

[1] Rakesh Agrawal, Alexander Borgida, and H. V. Jagadish. Efficient Management of Transitive Relationships in Large Data and Knowledge Bases. In *Proc. ACM-SIGMOD International Conference on Management of Data*, 1989.

[2] Daniel G. Bobrow, Linda G. DeMichiel, Richard P. Gabriel, Sonya Keene, Gregor Kiczales, and David A. Moon. Common Lisp Object System Specification. *SIGPLAN Notices*, Vol. 23, special issue, Sept. 1988.

[3] Daniel G. Bobrow, Kenneth Kahn, Gregor Kiczales, Larry Masinter, Mark Stefik, and Frank Zdybel. CommonLoops: Merging Lisp and Object-Oriented Programming. In *Proc. Conference on Object-Oriented Programming Systems, Languages, and Applications*, 1986.

[4] Luca Cardelli and Peter Wegner. On Understanding Types, Data Abstraction, and Poly-

morphism. *ACM Computing Surveys*, 17(4), December 1985.

[5] Linda G. DeMichiel and Richard P. Gabriel. The Common Lisp Object System: An Overview. In *Proc. European Conference on Object-Oriented Programming*, 1987.

[6] Patrick H. Dussud, TICLOS: An Implementation of CLOS for the Explorer Family. In *Proc. Conference on Object-Oriented Programming Systems, Languages, and Applications*, 1989.

[7] Margaret A. Ellis and Bjarne Stroustrup. *Annotated C++ Reference Manual.* Addison-Wesley, 1990.

[8] Daniel H. H. Ingalls. A Simple Technique for Handling Multiple Polymorphism. In *Proc. Conference on Object-Oriented Programming Systems, Languages, and Applications*, 1986.

[9] Gregor Kiczales and Luis Rodriguez. Efficient Method Dispatch in PCL. In *Proc. Conference on Lisp and Functional Programming*, 1990.

[10] C. Lécluse and P. Richard. Manipulation of Structured Values in Object-Oriented Databases. In *Proc. Second International Workshop on Database Programming Languages*, 1989.

[11] Warwick B. Mugridge, John Hamer, John G. Hosking. Multi-Methods in a Statically-Typed Programming Language. Report No. 50, Department of Computer Science, University of Auckland, 1991. (To appear in *Proc. European Conference on Object-Oriented Programming*, 1991.)

[12] Bjarne Stroustrup. *The C++ Programming Language.* Addison-Wesley Publishing Co., 1987.

[13] R. Tarjan. Depth-first Search and Linear Graph Algorithms. *SIAM Journal on Computing*, 1(2), 1972.