User-Defined Behavior of Generic Functions

William Kent

Database Technology Department, Hewlett-Packard Laboratories
1501 Page Mill Road, Palo Alto, California 94303 USA
kent@hplabs.hp.com

1 Introduction

1.1 Purpose

Overloaded functions (polymorphic operations) are a mainstay of object orientation. A "generic function" might be thought of as the equivalence class of functions having the same name. Little attention is paid to the generic functions themselves in most object-oriented languages. Even in languages such as OSQL [3], generic functions are implicitly defined and managed behind the scenes by the system.

Specific functions having a given name imply the existence of a generic function with that name. Allowing users to explicitly define behavior of generic functions to override the behavior implicitly defined by the system permits:

- Different behaviors of generic functions for different argument types.
- Disambiguation of ambiguous calls to overloaded functions.
- Different result types for overloaded functions with the same name.
- Default values for functions not defined on certain types.
- A limited form of upward inheritance.
- Multi-function uniqueness and equivalence specifications.

This capability is being proposed as an extension to OSQL [3]. Besides their general utility, these features are particularly useful for multidatabase systems [Section 4].

1.2 Overview

A behavior may be specified for a generic function f with respect to a set of relevant types. Specifications may include a result type, a default result value, a disambiguation specification, and a uniqueness specification (these terms will be defined shortly). Different behaviors may be specified for different sets of relevant types. For example, they may have different result types; any function named f whose argument type is in a relevant type set must have the specified result type.

When f(x) is invoked:

- If it is unambiguous, then the appropriate specific function is invoked, without involving the generic function.
- If x is an instance of types in a relevant type set for the generic function f, but no specific f is defined on any type of x, then the generic function can define the default result.
- If f(x) is ambiguous, and the argument types of all the eligible specific functions are in a relevant type set for the generic function f, then the generic function can provide the disambiguating behavior.

- When none of these conditions are satisfied, the system default behavior for generic functions applies.

Another extension proposed below would allow automatic resolution of overload ambiguity when the eligible functions are consistent, i.e., all return the same value.

2 Specific Functions

A *specific function* has:

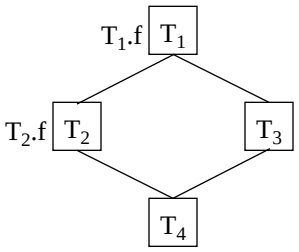
- A *simple name* f (which is also the name of the associated generic function).
- An argument type T, possibly an aggregate type.
- A result type, possibly an aggregate type.
- A behavior specification.
- A *specific name* composed of its simple name f and the name of its argument type T, which we will write as T.f. Specific names are unique over all functions.
- Possibly other characteristics, perhaps specified by keywords.

Inheritance behaves as follows:

- The function T_i.f is *known* for T_i.
- If the function T_i.f is known for T_j, then it is known for an immediate subtype T_k of T_j if there is no specific function T_k.f.

A subtype T_2 of T_1 is an *immediate subtype* of T_1 (and T_1 is an *immediate supertype* of T_2) if no supertype of T_2 is a subtype of T_1. An instance x of type T is an *immediate instance* of T (and T is an *immediate type* of x) if x is not an instance of any subtype of T.

In particular, both T_1.f and T_2.f are known for T_4 in the following case:



Unless otherwise defined for the generic function, all specific functions having the same simple name must have the same result type. (OSQL may allow overloaded functions defined on literals to have different result types from functions with the same name defined on surrogates. That's not particularly significant here.)

A function call may involve a simple (generic) or specific function name. If specific, the call invokes the behavior defined for the specific function.

Unless otherwise defined for the generic function, a generic function call f(x) is processed as follows:

1 The *eligible functions* are the specific functions T.f which are known for the immediate types of x.

2 If there is exactly one eligible function, invoke it and exit.

3 If there are no eligible functions:

  3a If strict type checking is in effect, then raise a type violation error and exit.

  3b Else return null (or empty) together with a warning condition (if warnings are supported) and exit.

4 If there is more than one eligible function:

  4a If all the non-null values are the same, then return that value (or null if all the values are null) and exit.

  4b Else raise an ambiguity error and exit.

Item 3b (relaxed type checking) is supported in OSQL.

In current OSQL, f(x) is considered ambiguous when there are several eligible functions — even if they have the same value. Item 4a is a proposed extension allowing automatic disambiguation based on *consistent values* of the eligible functions.

3 Generic Functions

A *generic function* has:

- A *generic name*.
- One or more sets of relevant types. For each set of relevant types, there may be
  - A result type.
  - A *default result specification*.
  - A *disambiguation specification*.
  - An optional uniqueness specification.

Behavior of a generic function may be defined with the following statement (the syntax is illustrative, in the style of OSQL [3]):

DEFINE GENERIC FUNCTION *generic-name*
[FOR *type-list*]
[RESULT_TYPE *result-type*]
[DEFAULT_VALUE [FOR *var-1* IS] *expr-1*]
[DISAMBIGUATE [FOR *var-1*] USING *expr-2* WITH {VALUE_BAG | FUNC_SET}
var-2]
[UNIQUE];

We will use the function name f for illustration, with f(x) being a generic function call. (Note: we don’t address the case where x is null.)

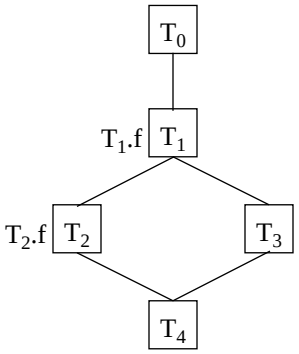
The statement may occur before any specific function named f is created, in order to provide control over result types.

The *relevant types* are those in the *type-list and their subtypes*. Types which become subtypes of these in the future will automatically be included. Different behaviors may be defined for different sets of relevant types for the same generic function f. A type may not be in more than one relevant type set for a given generic function. If *type-list* is omitted, then the relevant type set contains all types.

Specified behaviors are associated with a particular generic function and relevant type set.

If *result_type* is specified, then any specific function named f whose argument type is in the relevant set must have the specified result type. Specific functions named f may have different result types if their argument types are in different relevant type sets with associated result-type specifications. Specific functions named f whose argument types are not in any relevant type set having a result-type specification must all have the same result type; this is the default result type.

The DEFAULT_VALUE clause has the effect of making a function f defined for all the relevant types. If the types of x are in exactly one relevant type set for f having a DEFAULT_VALUE specification, and there is no specific function T.f known on any type T of x, then *expr-1* defines the value of f(x). If *var-1* is provided, it is bound to the value of x, and it may be used in *expr-1*. In the following example, if T_0 is in the *type-list* but there is no specific function T_0.f, then *expr-1* defines the value of f for immediate instances of T_0. This can be considered a limited form of *upward inheritance*.



The DISAMBIGUATE clause specifies action to be taken when a generic function call f(x) is ambiguous, and the argument types of all the eligible functions are in the relevant type set. The value of *expr-2* is returned; it must yield a value whose type is the specified or default *result-type*. The variables *var-1* and *var-2* may be used in *expr-2*. If *var-1* is provided, it is bound to the value of x. If VALUE_BAG is specified, then *var-2* is bound to the bag of non-null results of the evaluated eligible functions. If FUNC_SET is specified, then *var-2* is bound to the set of unevaluated eligible functions, allowing *expr-2* to reason over these functions and select the ones to be invoked.

The UNIQUE keyword signifies that the generic function must be unique-valued for all instances of all its relevant types. It means that $T_i.f(x)=T_j.f(y) \Rightarrow x=y$ if T_i and T_j are relevant types. Thus, for example, if a generic function named SSN ("social security number") was defined to be unique over a set of types, then distinct instances of relevant types may not have the same value of SSN,

even via different specific functions named SSN. In OSQL, without this extension, uniqueness can only be specified within an individual specific function.

Section 4 provides other examples.

The processing of a generic function call f(x) is now extended as follows:

1 The *eligible functions* are the specific functions T.f which are known for the immediate types of x.

2 If there is exactly one eligible function, then invoke it and exit.

3 If there are no eligible functions:

3a <u>If the types of x are in exactly one relevant type set for generic function f which has a specified result value, then return that value and exit.</u>

3b If strict type checking is in effect, then raise a type violation error and exit.

3c Else return null (or empty) together with a warning condition (if warnings are supported) and exit.

4 If there is more than one eligible function:

4a <u>If the argument types of all the eligible functions are in exactly one relevant type set for generic function f which has a disambiguation specification, then use that disambiguation specification and exit.</u>

4b If all the non-null values are the same, then return that value (or null if all the values are null) and exit.

4c Else raise an ambiguity error and exit.

As before, the non-underlined portions of items 3 and 4 characterize the default behavior of the generic function.

4 Application to Multidatabase Systems

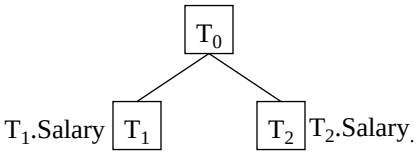
Multidatabase systems such as Pegasus and Multibase [1, 2] include the ability to reconcile inconsistent data from different data sources, and also to treat things imported from different data sources as the same object. These things can be done with user-defined generic function behavior.

The general approach to integration is to import data from different sources as distinct imported types, and then to provide correlation mechanisms across the imported types.

<u>4.1 Function Merging for Inconsistent Data</u>

A typical situation here is that data from two sources may be inconsistent, e.g., Salary(x) in one source differs from Wages(x) in another. If we impose the not unreasonable requirement that “semantically equivalent” functions should have the same name, even by renaming if necessary, then the disambiguation behavior of generic functions can be used to correlate their values.

Suppose we have the following schema after renaming, where T_1 and T_2 are imported types:



Then

```
DEFINE GENERIC FUNCTION Salary
FOR T0
RESULT_TYPE Number
DEFAULT_VALUE Return(0)
DISAMBIGUATE USING Average(sal_bag) WITH VALUE_BAG sal_bag;
```

defines the generic function behavior for Salary(x) which would be applicable if x is an instance of T_0 or any of its subtypes. Any Salary function defined on T_0 or any of its subtypes must have result type Number. When Salary(x) is invoked:

- If x is an instance of T_1 only or T_2 only, then T_1.Salary(x) or T_2.Salary(x) is invoked without even involving the generic function.
- If x is an immediate instance of T_0, or an instance of any subtype on which Salary is not known, then DEFAULT_VALUE defines the value of Salary(x) to be 0.
- If x is an instance of T_1 and T_2, then the DISAMBIGUATE clause indicates that the average of the salaries is to be returned. The variable sal_bag will contain the bag of non-null results of evaluating T_1.Salary(x) and T_2.Salary(x), which in this case will be passed to the Average function.

If the behavior was defined as

```
DEFINE GENERIC FUNCTION Salary
FOR T0
RESULT_TYPE Number
DEFAULT_VALUE Return(0)
DISAMBIGUATE FOR x USING Best_sal(x,funcs) WITH FUNC_SET funcs;
```

then, if x belongs to both T_1 and T_2, the functions T_1.Salary(x) and T_2.Salary(x) will not be evaluated. The variable funcs will contain the set of unevaluated functions T_1.Salary and T_2.Salary, which in this case will be passed to the Best_sal function along with x. That function might, for example, consider T_1 to always be more reliable, and only evaluate and return T_1.Salary(x). It might also do more complex reasoning, such as dealing with null values.

If no disambiguation was defined, then the default behavior in case of ambiguity under the present proposal would be to return the consistent value if T_1.Salary(x)=T_2.Salary(x), and raise an ambiguity error otherwise.

This approach to correlating imported data has several benefits:

- It takes advantage of a general-purpose facility for disambiguating overloaded functions.
- Nothing has to be specified for the cases where there is no conflict, e.g., x belongs to only one of the subtypes.
- It is readily extensible. When a new type is imported, it is only necessary to insure that the Salary function has that name, and to make the new type a subtype of T_0. The mechanism described here will automatically be applicable.

Accidental conflicts with other unrelated functions which might also be named "Salary" can be avoided by

- Renaming it to a different name, or
- Making sure its argument type is not a subtype of T_0.

4.2 Merging Equivalent Objects

Multidatabase systems provide the ability to treat objects imported from different external data sources as being the same object, based on some criterion such as social security number. The general mechanism is to provide equivalence specifications across the imported types [2]. Unique-valued generic functions can simplify their specification.

A simple equivalence specification might take the form

$$T_1.\text{SocSecNum}(x)=T_2.\text{SSN}(y) \Rightarrow x=y,$$

implying that x and y are to be treated as the same object if $T_1.\text{SocSecNum}(x)=T_2.\text{SSN}(y)$. Generic functions can become involved if we again adopt the same-name assumption for semantically equivalent functions. If the functions are all named SSN, and the generic function named SSN is defined to be unique for T_1 and T_2, this would imply

$$T_1.\text{SSN}(x)=T_2.\text{SSN}(y) \Rightarrow x=y.$$

More generally, we would have

$$T_i.\text{SSN}(x)=T_j.\text{SSN}(y) \Rightarrow x=y$$

whenever T_i and T_j were relevant. Thus a single generic function behavior definition can imply equivalence specifications between many pairs of types.

The equivalence holds even when T_i and T_j are the same. Thus if two people in the same imported type somehow had the same social security number, they would be treated as the same person. (A more plausible example might arise when courses using the same textbook are to be treated as the same course.) Of course, the earlier treatment of function merging does not work for such intra-type equivalence. Such intra-type equivalence is beyond the scope of the present paper.

We need to distinguish between two responses to "violations" of uniqueness:

- Rejecting it as an error, e.g., trying to give the same social security number to two people known to be distinct.
- Treating the offenders as being one and the same object.

This can be distinguished on the basis of a "view" concept:

- If two objects are known to be distinct in the current view, then any attempt to modify their properties in a way that would violate uniqueness is rejected as an error.
- If one or both of the objects are "imported" from some other view with matching values of unique properties, then treat them as a single object.

Equivalence based on conjunction of multiple properties, e.g., Nationality and PassportNumber, can be handled by defining another function on each of the imported types as the concatenation of Nationality and PassportNumber, e.g.,

Ident(x) ::= <Nationality(x),PassportNumber(x)>,

and then defining the generic function Ident to be unique.

Equivalence based on disjunction of multiple properties, e.g., social security number or passport number, can be handled by separately defining a unique-valued generic function for each. (Cycles of such equivalence specifications can also cause intra-type equivalence. A person in one type might have a social security number corresponding to one person in another type and a passport number corresponding to a different person in that type.)

As before, this approach is readily extensible. New imported types can be accommodated by appropriately naming the functions and making the types relevant to the generic type.

5 Conclusions

Allowing users to explicitly define behavior of generic functions permits:

- Disambiguation of ambiguous generic calls to overloaded functions.
- Different behaviors of generic functions for different argument types.
- Different result types for overloaded functions with the same name.
- Default values for functions not defined on certain types.
- A limited form of upward inheritance.
- Multi-function uniqueness and equivalence specifications.

This capability is being proposed as an extension to OSQL [3].

References

1 R. Ahmed, P. DeSmedt, W. Du, W. Kent, M. Ketabchi, W. Litwin, A. Rafii, M.-C. Shan, "The Pegasus Heterogeneous Multidatabase System", IEEE Computer, December 1991.

2 Umeshwar Dayal and Hai-Yann Hwang, "View Definition and Generalization for Database Integration in a Multidatabase System", IEEE Trans. on Software Engineering, SE-10(6), Nov. 1984, pp. 628-645.

3 Dan Fishman, et al, "Overview of the Iris DBMS", *Object-Oriented Concepts, Databases, and Applications*, Kim and Lochovsky, eds, Addison-Wesley, 1989.

User-Defined Behavior of Generic Functions

William Kent

Database Technology Department, Hewlett-Packard Laboratories
1501 Page Mill Road, Palo Alto, California 94303 USA
kent@hplabs.hp.com

857-8723

Keywords

object models, overloaded functions, overloaded operators, polymorphism, generic functions, generic operators, behavior specification, interoperability, multi-database

Abstract

Specific functions having a given name imply the existence of a generic function with that name. Allowing users to explicitly define behavior of generic functions to override the behavior implicitly defined by the system permits:

- Different behaviors of generic functions for different argument types.
- Disambiguation of ambiguous calls to overloaded functions.
- Different result types for overloaded functions with the same name.
- Default values for functions not defined on certain types.
- A limited form of upward inheritance.
- Multi-function uniqueness and equivalence specifications.

This capability is being proposed as an extension to OSQL. Besides their general utility, these features are particularly useful for multidatabase systems.