Adding Generic Functions to Scheme

Thant Tessman
Walt Disney Imagineering
thant@disney.com

Abstract

A generic function is a set of methods, all of which have equivalent semantics but each of which apply to different domains. This document describes how to augment the Scheme programming language with the capability to define multi-method generic functions. Also described are optimization issues and areas of potentially fruitful future exploration.

Introduction

A *domain* is the set of values over which a function is defined. A *generic* domain is a set of values of more than one type. An *ad hoc* generic domain has subdomains that are related by their semantics rather than by having some common structure. A *generic function* is a single abstract operation that is defined over an ad hoc generic domain. To handle ad hoc generic domains, a single generic function can consist of several independent blocks of code with each block designed to handle different arguments, but with all blocks sharing a common name [1].

For example, many programming languages provide more than one way to represent numbers, and the code to add integers is different from the code to add reals, which is in turn different from the code to add complex numbers. However, every version of the addition function is usually represented by a single symbol: "+". The compiler decides which version is appropriate by examining the arguments provided.

When a generic function is defined by more than one body of code, each of the separately defined bodies is called a *method*. When more than one method is bound to a single name in the same scope, the name is said to be *overloaded*. The Scheme programming language does not by itself provide the capability for the programmer to define generic functions as multiple methods. However, Scheme is powerful and flexible enough that it is possible to add this capability with the help of a syntactic extension mechanism. The generic function mechanism described here has been implemented in both Chez Scheme [2] and MacScheme [3] with the `extend-syntax` macro facility developed by Eugene Kohlbecker [4] and described in [2].

Generic Method Dispatching

The process of deciding which method to use in a given context is called *dispatching*. The dispatcher uses information about the function arguments to choose an appropriate method. This can happen at compile-time for a statically-typed language such as C, or while a program is running for a latently-typed language such as Scheme.

Object-oriented programming languages allow the user to define methods specific to a given class. When a function is applied to an object, the dispatcher selects a method based on the object's class. Aficionados of object-oriented methodology find it useful to think of an object's methods as though they were contained "within" the object itself, and more traditional object-oriented programming languages (e.g. Smalltalk [5]) support dispatching only on a single object's type.

Other languages (for example CLOS [6], Dylan [7], and even C++ [8]) have generalized the concept and dispatch on the types of more than one argument. This is called *multi-method* dispatching. The generic function mechanism described here implements multi-method dispatching. While it is not as feature-laden as CLOS, it is simple and efficient and has proven in practice to be quite useful.

Generic Functions

How to use generic functions is best illustrated with a simple (but contrived) example:

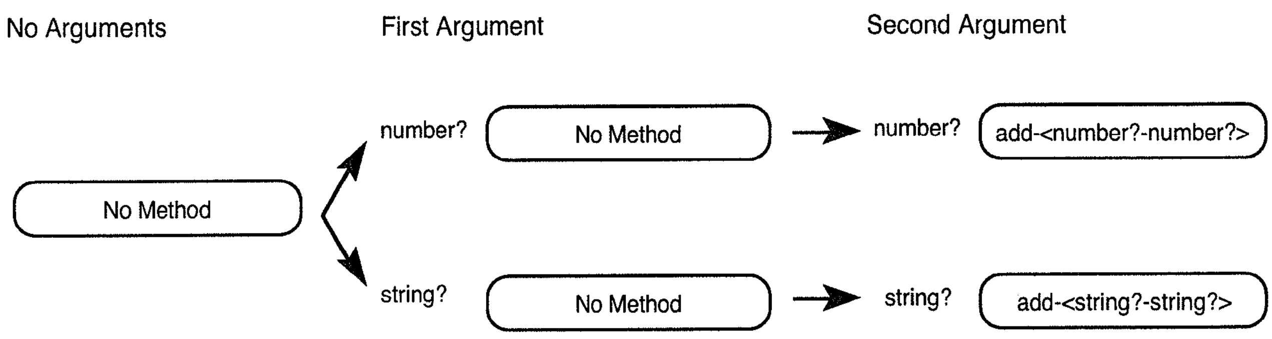
```
(define-generic (add (number? a) (number? b))
  (+ a b))

(define-generic (add (string? a) (string? b))
  (string-append a b))

(add 3 4) → 7
(add "auto" "mobile") → "automobile"
```

The `define-generic` macro creates a scheme procedure just as `define` would. However, it does not bind the procedure to the function identifier. Instead it inserts the procedure into a database of methods, associating it with the function name and the specified list of type predicates. The macro then uses the information in the database to build the method dispatcher for that function. It is the dispatcher that is bound to the function identifier.

Method Database

Each method of a generic function is distinguished by a unique list of type predicates. This list of type predicates comprises a method's signature. One possible way to implement generic functions might be to store a list of method signatures for each function. The dispatcher could take the arguments supplied to the function and go through the list of signatures, first checking to see if the number of arguments supplied to the function matches the number of arguments in the signature, and then applying each type predicate to each of the arguments. If all of the type predicates return true, the signature is considered a match, and the method corresponding to that signature is applied to the arguments and the result returned. If not, the dispatcher continues through the list. If the arguments don't survive the application of any signatures, an error is signaled.

A much more efficient way to organize the database of methods is as a tree. The depth of the tree corresponds to the number of arguments. The branches of a node at depth *n* correspond to the type predicates to be applied to the n^{th} argument. (For example, if a method of zero arguments was defined for a given generic function, it would be stored in the root node of the tree.) Dispatching is the process of matching the n^{th} argument against the predicates at the n^{th} level of the tree. If a match for that argument is found, the process continues with the n+1 argument down the subtree under the matching branch. When there are no more arguments, the method at that node (if there is one) is applied to the full set of arguments. If there is no method at the node the dispatcher winds up on when there are no more arguments left, or if one of the arguments doesn't match any of the predicates at a given node, an error is signaled. This is a diagram of the tree produced by the previous example:



Subtypes

There is a subtle problem with what has been described so far. A generic function might contain methods that are defined for overlapping subdomains. When the domain of one method is a subdomain of another method, the intent of the programmer is almost always that the more specific method is to take precedence over the less-specific. In the implementation of multi-method dispatching as it has been described so far, there is no guarantee that this will be the case.

For example, Scheme provides for several representations of numbers organized in a tower of overlapping subtypes. These types are: *number*, *complex*, *real*, *rational*, and *integer* [9]. All integers are rational. Anywhere a rational number is expected, an integer may be used. The Scheme type predicate `rational?` will always return true when given an integer. Also, all rational numbers are real numbers. And so on.

Suppose two generic methods are defined. One takes a real number and the other takes an integer. Depending on the order that the methods were defined, the one that takes a real might be stored in the method database such that a match for it is checked before a match for the other. The method taking a real could effectively hide the method taking an integer because any argument that would satisfy the signature for the integer method would be caught first by the signature for the real method.

The solution is to sort the type predicates by their relationship in the type hierarchy. This requires that the procedure that inserts methods into the method database know the type hierarchy. To make this possible, two more procedures are provided; one to declare a subtype relationship (called `is—a`), and one to query for a subtype relationship (called `is—a?`). The subtype relationships for the Scheme built-in values are declared in the following code:

```
(is-a 'integer? 'rational?)
(is-a 'rational? 'real?)
(is-a 'real? 'complex?)
(is-a 'complex? 'number?)

(is-a 'list? 'pair?)

(is-a? 'integer? 'real?)  → #t
(is-a? 'real? 'integer?)  → #f
(is-a? 'pair? 'string?)  → #f
```

(Why the predicates are specified as symbols will be explained later.) Our implementation allows multiple subtypes of a given type to be declared, but it currently doesn't allow a type to be declared the subtype of more than one supertype. In other words, the implementation maintains a type hierarchy, but not a type heterarchy. (A heterarchy is a directed acyclic graph. A type heterarchy would be useful for, among other things, expressing multiple inheritance.) There is no theoretical reason why a type heterarchy won't work, but the `is-a` declaration does need to guarantee that no circularities in the type system are created. The `is-a?` procedure is used by the procedure that inserts methods into the method database to sort the predicate branches from most specific to least specific.

One more point: The search down the branches of the method signature tree needs to be able to backtrack down branches higher in the tree if a search lower in the tree fails.

```
(define-generic (process (integer? a) (integer? b))
  "processing integers")

(define-generic (process (real? a) (real? b))
  "processing reals")

(process 2 4)       → "processing integers"
(process 2.3 4.5)  → "processing reals"
(process 2.3 4)    → "processing reals"
(process 2 4.5)    → "processing reals"
```

First Match versus Best Match

Given subtyping, it is possible for more than one method of a generic function to match a provided set of arguments. The dispatcher described will select the first match it finds. It will favor matching a specific type over a general type early in the argument list over a match that may contain several better matches later in the argument list. This behavior is subtly different than that of other languages, but it has the definite advantage of being more efficient and simpler to implement. Also, it is worth noting that it (serendipitously) models object-oriented methodology better than 'straight' best match. Methods are typically written in a way that considers the first argument to be the object to which the the method belongs. First match gives the first argument the greatest weight in determining which method is called.

Implementation

This is the method signature tree for the add generic function given the example presented earlier:

```
(#f (number? #f (number? add<number?-number?>))
    (string? #f (string? add<string?-string?>)))
```

This is the dispatch code generated by the define-generic macro:

```
(define (add . args)
  (let ((result #f))
    (if (if (null? args)
            #f
            (let ((G165 (car args))
                  (args (cdr args)))
              (or (and (number? G165)
                       (if (null? args)
                           #f
                           (let ((G167 (car args))
                                 (args (cdr args)))
                             (or (and (number? G167)
                                      (if (null? args)
                                          (begin
                                           (set! result
                                                 (add<number?-number?> G165 G167))
                                           #t)
                                          #f))))))
                  (and (string? G165)
                       (if (null? args)
                           #f
                           (let ((G166 (car args))
                                 (args (cdr args)))
                             (or (and (string? G166)
                                      (if (null? args)
                                          (begin
                                           (set! result
                                                 (add<string?-string?> G165 G166))
                                           #t)
                                          #f)))))))))
        result
        (generic-error 'add args))))
```

The code produced by the define-generic macro wasn't designed to be readable, and the predicate tree isn't very tree-shaped in this example, but there are still things worth noticing. First, the or expressions map across branches of the method predicate tree, and the and expressions map down the predicate tree. Second, the procedure looks much bigger than it really is because there are several or and and expressions that contain only a single argument.

The or clauses serve as the backtracking mechanism. If at any point in the tree a branch fails (because a subtree provides no match), the or expression will move on to the next branch. If no branches provide a match, the node returns #f to its parent, which in turn tries the next branch.

Optimization

Our first implementation stored the procedure to be used as a method directly into the method database. The dispatcher was a procedure that took the name of the generic function and a set of arguments and used them to navigate the method database to the appropriate procedure. The define-generic macro bound the function name to a call to the dispatcher. This implementation was concise, but it turned out to be very slow.

A significant improvement in performance was achieved by using the information in the database to produce code that would dispatch directly for a given function. (An example of this is what was shown previously.) However, with this technique, it proves to be simpler to store the *name* of a method procedure in the the method database, or the code for the method itself. This is also true for the predicates. The dispatcher uses the name of the method when generating the dispactch code. (This is why the is-a and is-a? procedures took symbols.)

We have found it convenient to build a (hopefully unique) identifier out of the function name and the predicate names, and bind the method to that identifier. It is the name of the method that is stored in the method database (as shown previously). One advantage of doing it this way is that it provides the capability to 'hand-dispatch' in speed-critical sections of code when the argument types are known.

Scheme facilitates incremental development by allowing top-level definitions to be changed in such a way that any function that refers to a top-level value always gets the latest value even if that value is changed. Since generic functions are top-level values, they are automatically extended for every function that uses them any time a new generic method regenerates the dispatch code for that function.

The technique that produced our second biggest leap in performance was taking advantage of Chez Scheme's case-lambda feature. It allows the programmer to write lambdas that dispatch on the number of arguments supplied. This made the (function . arguments) form in the dispatching code unnecessary. Execution time was reduced not only because it was no longer necessary to check for the end of the argument list, but also because argument values were bound directly to identifiers once up front. Unfortunately, case-lambda is not a standard feature of Scheme.

There is one last technique we used to improve performance. When there is only one method for a given function, the generic-dispatch macro doesn't bother to create the code for the dispatcher. Instead, it binds the method directly to the function name. It is worth noting that the dispatcher is no longer doing type-checking on the arguments. This may or may not matter, but programmers using this generic dispatch system might be more willing to use define-generic over define for writing single functions if they know that the resulting code will be just as efficient. (Plus, the method database can be used as online documentation, and using define-generic over define automatically documents the function's signature.)

Object Systems

The reader may have noticed the conspicuous absence in this document of any mention of an object system other than the Scheme built-in types. This should not be taken to imply that this system is only appropriate for Scheme objects. This system is extremely general. It assumes nothing about the implementation of objects. The programmer merely needs to create type predicates and notify the generic dispatch generator of subtype relationships. And this task can be automated with an object system. In fact, this system can be (and has been) used with several different object systems simultaneously, even within the signature of a single method. And, of course, users can program predicates that check not just for type, but for specific values.

Other Issues and Future Development

Limitations Besides the limitations discussed previously, our implementation currently doesn't handle methods that take a variable number of arguments. Also, it doesn't handle multiple return values (to be described in the successor to R4RS [9]).

Scoped Methods The implementation described assumes that generic methods will only be defined in the top-level environment. I've recently implemented a `let-generic` form that allows the user to define methods that are confined to a local scope. It builds a dispatcher from the signature specified in the form, and it replaces the call to the error function with a call to any version of the function that might be defined in an outside scope.

Type Coercion Type predicates could be made to return not false or true, but false or a value of the proper type. This would give predicates the opportunity to cast other types to the proper type. (I suspect type coercion and multiple supertypes are related in some fundamental way, but I haven't thought it out yet.)

Type Analysis The most intriguing possibility for future development is to take advantage of the fact that the arguments passed into a method are of known type. It should be possible to use this knowledge to eliminate redundant type-checks in the dispatching of other generic functions within the body of the method. The `define-generic` macro could check to make sure the arguments aren't modified, and then, if one or more of the arguments was passed to another generic function, it would expand the dispatcher code for that generic function and replace any redundant type-checks with the value `#t`.

Conclusion

Generic functions facilitate programming methodologies (like OO) usually associated with languages other than Scheme. Generic functions can be added to Scheme in a portable, efficient, and non-invasive 'Scheme-friendly' way using syntactic extension. When implementing generic functions, the existence of subtype relationships among different domains must not be ignored.

Acknowledgements

Some of the ideas presented were inspired by conversations with Scott Watson.

References

[1] Alice E. Fischer and Frances S. Grodzinsky: *The Anatomy of Programming Languages*
Prentice Hall, Englewood Cliffs, New Jersey, 1993.

[2] R. Kent Dybvig: *The Scheme Programming Language*
Prentice Hall, Englewood Cliffs, New Jersey, 1987.

[3] Lightship Software: *MacScheme Manual and Software*
MIT Press, Cambridge, Massachusetts, 1990.

[4] Eugene E. Kohlbecker: *Syntactic Extensions in the Programming Language Lisp*
Ph.D. Thesis, Indiana University, 1986.

[5] Goldberg, A. and D. Robson: *Smalltalk-80: The Language and Its Implementation*
Addison-Wesley, Reading, Massachusetts, 1983.

[6] Sonya E. Keene: *Object-Oriented Programming in Common Lisp: A Programmer's Guide to CLOS*
Addison-Wesley, Reading, Massachusetts, 1989.

[7] Apple Computer Eastern Research and Technology: *Dylan: An Object-Oriented Dynamic Language*
Apple Computer, Cupertino, California, April 1992.

[8] Ellis, Margaret A. and Stroustrup, Bjarn: *The Annotated C++ Reference Manual*
Addison-Wesley, Reading, Massachusetts, 1990.

[9] William Clinger and Jonathan Rees, eds.: *The Revised^4 Report on the Algorithmic Language Scheme.*
Lisp Pointers IV, 3, 1992, 1-55.