Reflectively Optimizing Generic Function Dispatch

Harley Davis
Ilog, S.A.
2, avenue Galliéni
94253, Gentilly, France

ECOOP 1992 Reflective Workshop Position Paper

April 18, 1992

1 Introduction

In ΤΕΛΟΣ[1] and similar languages, a *generic function* is a function whose executed code depends on the class of its arguments. Each piece of executed code is represented by a *method*, which stores, among other things, a list of argument classes for which the method is applicable and the code to be executed. Abstractly, when a generic function is called, it must determine the complete list of all methods applicable for the arguments, sorted from most to least specific. The first is called; the others are called sequentially by calls to `call-next-method` within a method body.

For generic functions to compete with normal functions, this process must be optimized as much as possible. This paper presents several generic function call optimization techniques based on reflective examination of the generic function's environment. By *reflective*, we simply mean that the relevent aspects of the environment are explicitly represented and a standard protocol exists to examine them. None of these techniques require special compiler support. Most of them are implemented in LE-LISP version 16, a commercial product sold by Ilog, S.A.

Each generic function has associated with it a so-called *discriminating function*. This function is called when the generic function is called, and is responsible for doing all the work of the generic function — computing the applicable methods for the given set of arguments, and calling the first such method.

A naive approach would have a single such discriminating function, shared by all generic functions, which took a list of arguments and the generic function object itself, and computed and called the applicable method each time.

Typical intermediate approaches use a single discrimination function combined with a method cache to increase its performance.

However, a more sophisticated approach generates a discriminating function specific to its generic function by examining the structure of the generic function, its methods, and the classes on which those methods are specialized. Such a discriminating function, with its specific knowledge, can avoid much of the work of the general version. In this paper, we discuss the various dimensions of information about the generic function which can be used to generate optimized discriminating functions.

---

[1] ΤΕΛΟΣ is the object-oriented component of EULISP and LE-LISP version 16

2 Basic Implementation Assumptions

In this paper, we make several assumptions about the nature of the underlying implementation. None are based on particular machine or compiler architectures, nor do they affect the principle observations made.

- Associated with each class is an integral *class number* uniquely identifying that class. This number could simply be the address of the class object, or an associated number allocated when the class is created.
- Some optimizations rely on the fact that some classes of object have an especially efficient way to determine their class's class number. This is typically the case for tagged or immediate objects such as fixnums, characters, and, in some implementations, cons cells. If this is not the case, these optimizations will not apply.
- A basic cache mechanism based on hashing techniques is available for use. The cache size and format as well as the exact hashing function should be specifiable parameters of this facility.

3 When to generate the discriminating function

Clearly, the more information available when the discriminating function is generated, the more optimized it can be. In ΤΕΛΟΣ, the goal of complete knowledge is stymied because the state of the generic function can change dynamically over its lifetime. Thus, a discriminating function can only count on information which will be valid throughout the discriminating function's existence.

To increase the amount of knowledge which the discriminating function generation routine can use, we can regenerate it whenever the state of the generic function changes. (In a real implementation, the function will in fact be regenerated lazily, at the first call after the generic function changes state.)

We must thus determine what generic function state counts as "interesting"; this state will be exactly that useful in optimizing the discriminating function.

4 Dimensions of Discriminating Function Optimization

We have identified the following sources of knowledge which can be used to optimize the discrimination:

4.1 Number of potential effective methods

One of the most potentially useful sources of information is the maximal number of effective methods which the generic function might have to supply. An *effective method* represents the set of applicable methods for a particular argument signature. If this number is very small, especially fast discrimination techniques can be used. This calculation is based on the following factors:

- The number and identity of methods on the generic function. Call this number m. Call the methods themselves $M_i, i \in \{1 \ldots m\}$.
- The number and identity of discriminated arguments in the generic function. Call the number d. We assume this is fixed over all the methods, although theoretically it could be different for each method. We also assume for simplicity that the discriminated arguments are always the first d arguments of the generic function, although this too may not be the case.
- For each method and discriminated argument, the specializing class for that method and argument. Call these classes $C_{i,j}, i \in \{1 \ldots m\}, j \in \{1 \ldots d\}$.
- The size of the inheritance graph below each of the specializers. In other words, the total of number of direct and indirect subclasses of each specializer (including the specializers themselves). Call these numbers $S(C_{i,j}), i \in \{1 \ldots m\}, j \in \{1 \ldots d\}$.

Now, the total number of potential effective methods is:

$$\sum_{i=1}^{m} \prod_{j=1}^{d} S(C_{i,j})$$

If this number turns out to be exactly 1, we can avoid method caching techniques, and simply compare the classes of the specializers to those of the actual arguments, and transfer to the method if they are identical.

In any case, we can use this number to determine the maximum size of the method cache. Combined with d, this can be used to determine the optimal number of bits to extract from each argument class for determining the initial cache key.

More generally, given all the potential specializing classes, we can, in some cases, generate a rapid and space-efficient perfect hashing function for method cache probes.

4.2 Specializing Classes

Some classes — such as fix and character — may have a more effective type test than the extraction and comparison of the class number from the heap. If we can determine that methods on the generic function only specialize on these classes, we can again avoid a method cache and call efficient class-specific predicates on the actual arguments.

A simple, similar optimization is available if the only method on a generic function specializes on the root of the class hierarchy. In ΤΕΛΟΣ this class is called `object`; in CLOS, `t`. In this case, the single method is universally applicable and its method function may be used instead of the discriminating function, rendering these generic functions exactly or nearly as efficient as normal functions.

Although simple, this is a particularly important optimization because, modulo space considerations, it allows users to define many of their functions as generic without worrying about efficiency if they are not specialized. This is an implicit encouragement to provide more open software architectures with specializable protocols.

4.3 Known Classes of Methods

Some kinds of methods are generated automatically, and their bodies are thus known to the system. This is the case for slot accessor methods, for example.[2] If a generic function contains only accessor methods, it is not necessary to transfer to the method function; we can inline the method body in the discrimination function itself. Furthermore, in the case of accessors, if we can guarantee that the user has not intervened in the slot access protocol for a certain accessor, we can actually use the optimized access form directly in the discriminating function, rendering the most common cases of generic accessors almost as efficient as non-generic accessors.

ΤΕΛΟΣ also provides an explicit declaration for predicate generic functions, which return `true` when their single argument is an instance of a subclass of a given class, and `false` otherwise. In this case, the method body is trivial, and an actual transfer of control to the method body is unwarranted. Instead of storing an effective method or list of applicable methods in a predicate generic function's method cache, simply the return value need be stored. The discriminating function need only return this value.

Generally, more declarative information supplied by the programmer concerning the intention of her entities makes it more likely that the system will perform certain optimizations. ΤΕΛΟΣ favors explicit declarations for as many aspects of the system as possible not only to help the human reader of a program, but to encourage the generation of this kind of optimization. Many of the same optimizations could be generated without these declarations — for example, more general optimizations similar to those described above might determine if all of the methods in a generic had a constant method body or if they all call `slot-value`. However, optimizations of this kind are considerably more difficult to implement as they rely on code analysis. It is thus less likely that some implementation will do them, and will probably make such implementations larger.

4.4 Number of Generic Function Arguments

Another dimension to be considered is the total arity of the generic function, independent of the number of actually specialized arguments. It will in general be faster to generate an arity-specific function rather than an n-ary function which is `apply`ed to a list of arguments. While this does not substantially increase the amount of information usable in discriminating function generation, it complicates it since any particular optimization must tediously generate cases for any number of arguments (up to some surrender number, at which point an n-ary function must simply be generated.)

5 A Framework for Generating Discrimination Functions

All of the knowledge sources listed above rely on the following aspects of the generic function: Its methods, its arity, the number of discriminating arguments, and the specializing classes and their subclasses. Thus, to guarantee that all this information can be used in our optimized discriminating

[2] A *slot accessor* is an automatically generated generic function which retrieves or sets the value of a field in a class.

functions, the discriminating functions must be regenerated whenever any of this information changes. This happens in the following cases:

- When a method is added to or removed from the generic function.
- When a new subclass is added to a specializing class.
- When a specializing class is redefined. This might change the position of a slot in the class, forcing a regeneration of an optimized accessor discriminating function.

Of course, a real implementation will only note when these events occur by marking the generic function so that the discrimination function will be regenerated only when it is actually needed.

The final discriminating function will have several parts:

1. **Argument validation:** Is there some method applicable to the given arguments?
2. **Effective method determination:** If there are any applicable methods, which are they?
3. **Method application:** Finally, call the effective method with the given arguments.

Depending on the case, any of these steps might be elided or optimized. Most frequently, the argument validation phase will directly return the effective method. However, sometimes we can skip step two because we know the applicable method by other means (*eg*, there is only one of them). Other times, we can skip step three because we know exactly what the body of the method will be and can inline into the discriminating function. Additionally, the exact way to do step one is highly susceptible to optimization. It is there that the choice of method cache size, structure, and access function come into play.

6 Optimizations

The possible optimizations can be thought of as filling in holes in a matrix with the dimensions described above. It would be a generic function, in fact, except that much of the information is not class-based. Anyway, here we describe a few of these holes.

6.1 Single Method Generic Functions

The interesting fact here is that either the sole method is applicable, or none is. Also note that we need not worry about `call-next-method` in this case, since there will be no next methods.[3] We can divide this case into the following subcases:

[3]This is actually a special case of generic functions which do have a flat method structure: No methods are ever inherited. Depending on the implementation of `call-next-method`, this case may be independently optimizable, especially for the structure used to store the applicable methods.

6.1.1 Single Potential Method Generic Functions

In this case, the one method specializes only on leaf classes. We can close over the method function and the specializers' class numbers in the discriminating function, creating a poor man's method cache. The discrimination will simply compare the actual arguments' class numbers to those of the specializers, and if they are identical, branch to the method function.

6.1.2 Multiple Potential Method Generic Functions

Here, depending on the exact number of potential methods and the empirical cache performance time, it may still make sense to explicitly and sequentially test all possibilities rather than hash to a cache.

6.2 Optimized Caches

The general hashing algorithm will make some computation over the set of specialized argument class numbers, either masking and shifting parts of each or doing some multiplication and division. However, in some cases we can do better.

6.2.1 Linearized Class Numbers

We can make use of three empirical facts about closed object-oriented applications to generate further optimizations:

1. Classes allocated sequentially may also be assigned sequential class numbers.
2. Classes in an inheritance hierarchies are typically allocated together.
3. Methods on a single generic function tend to be written only for a particular class inheritance hierarchy.

We can use the resulting locality of class numbers within a generic function's methods to generate a better hashing algorithm if the class numbers of the specializers of all of the potential effective methods for a real method are linear. In this case, we close over the low and high bounds and the real method, and just do a bounds check on the class number of the actual specialized argument. This may or may not be useful if there is more than one actual method.

Clearly this optimization does not apply to open, extendable architectures. However, in our experience closed applications account for a significant percentage of object-oriented programs.[4] A final, delivered application may contain only closed subsystems.

[4]For example, a recent examination of the basic LE-LISP version 116 development environment revealed that 178 classes out of a total of 361 belong to closed hierarchies.

6.2.2 Distinguished Class Number Bit Patterns

If we can determine that some mask of all the potential specializers uniquely determines the applicable method, we can generate a perfect hashing function using that pattern. To be useful, the size of the cache must remain small under any such hashing function. This optimization may be the most difficult to generate.

6.3 Accessor Generic Functions

This case concerns generic functions which only have accessor methods on them. This case can be divided into subcases according to the knowledge we have about the access method, as well as being subject to the optimizations described above. The most general case would call the slot description reader and writer directly from the discrimination function. In other words, we do a case based analysis of the metaclass hypotheses of all the specializing classes and their subclasses, and based on this selection, choose an optimized access form directly in the discrimination function. A less grandiose implementation might simply check if there are only known and intercompatible metaclasses in the specialization hierarchy, and use built-in access forms for these, degenerating to an explicit call to the accessor otherwise.

If several accessor methods are attached to a generic function, we can still analyze whether all the methods would use the same access protocol, or if they share a common slot position, and generate a single inlined access form based on that information. This is relatively independent of the argument validation.

7 Development Environment Versus Runtime Environment

It is important to distinguish development needs from application runtime needs. Since the user of one is typically not the user of the other, the requirements differ.

In a dynamically adaptive framework, programmers who frequently redefine classes and methods will suffer severe performance penalties. Active developers do not need ultimate performance, but reasonable and reliable performance; only users in a relatively stable environment will appreciate the benefits of optimized discrimination.

LE-LISP version 16 provides a convenient way to distinguish these states: A module can be at one of several development levels, representing modules under active development, stabilized but open modules, and modules in closed applications. At the active development level, generic functions should use a single generic discrimination function which will not be regenerated during the generic function's lifetime. However, at the higher module development levels, the full optimized system should be used.

8 Conclusion

We have seen a series of generic function optimizations which can be performed at runtime to dynamically determine the best discrimination function for a particular system state. These

optimizations are made possible by the reflective, explicitly represented environment of TELOS. These techniques could also be applied to other reflective languages, such as CLOS or SMALLTALK.