ACM OOPS Messenger **2**, 4 (October 1991), 4-15.

CLOStrophobia[1]: Its Etiology and Treatment

Henry G. Baker

Nimble Computer Corporation, 16231 Meadow Ridge Way, Encino, CA 91436
(818) 986-1436 (818) 986-1360 FAX

This work was supported in part by the U.S. Department of Energy Contract No. DE-AC03-88ER80663

The Common Lisp Object System (CLOS) has received some praise and some criticism, both deserved. One of the most controversial features of standard CLOS is its linearly-ordered "class precedence list", which is used to linearly order the execution of its "combination" methods. In addition to the problems already known regarding the linear ordering of superclasses, we show that the standard CLOS class precedence ordering produces gratuitously complex and non-intuitive behavior. We then show that a slight modification of the standard ordering rules produces a linear ordering which can achieve most of the goals of CLOS more efficiently, and without impacting most programs. We describe a subset of CLOS called *Static* CLOS which preserves much of the praise due CLOS, while eliminating some of the criticism. Static CLOS is tuned for *delivery* of debugged code, rather than for prototype development. This "delivery" CLOS determines as many methods as possible at compile-time using "type inference" techniques. While these techniques generally result in faster-executing code, the space requirements can grow quite large. We argue that this space explosion can be partially ameliorated through the use of our modified class precedence ordering.

INTRODUCTION

CLOS—the Common Lisp Object System [Steele90,ch.28] [Moon89] [Dussud89]—is Common Lisp's entry into the object-oriented programming sweepstakes.[2] Because Lisp has traditionally been the domain for investigating some of the most complex software ideas, CLOS attempts to be the "biggest and baddest" object-oriented programming system, which means in this context that it has the most power and the most flexibility. It offers multiple inheritance, generic functions with the capability of dispatching on multiple arguments, "combination" methods, first-class classes, first-class generic functions, a dynamic class system and a dynamic system of generic function methods. Furthermore, its "meta-object protocol" [Kiczales90b] [des Rivières90] offers the option to replace substantial portions of CLOS internals, so that a wide variety of other object-oriented paradigms can be emulated. Unfortunately, CLOS has sacrificed "information hiding" on the altar of "reflection" and therefore CLOS provides only "translucent", rather than "opaque" abstract data types.[3]

The consensus seems to be building in the Computer Science community that object-oriented programming languages are a "good thing", and hence much effort has gone into the problem of implementing them efficiently on today's RISC architectures, e.g., [Borning81] [Ballard83,86] [Conroy83] [Hagmann83] [Deutsch84] [Suzuki84] [Atkinson86] [Johnson86] [Samples86] [Bush87] [Stroustrop87] [Ungar87] [Rose88] [Chambers89,90] [Dixon89] [Mellender89] [Kiczales90] [Cyphers90]. Unfortunately, object-oriented programming languages still suffer a significant performance penalty compared with procedure-oriented languages. While the price/performance of today's architectures is good enough that object-oriented programming is cost-effective, much can be gained by better object-oriented software architectures and smarter compilers. This paper is an attempt to trim some of the rough performance edges from the CLOS language, and to show how such languages can be more efficiently compiled.

---

[1]A pun on *claustrophobia*, a morbid dread of closed or narrow places. CLOS shows every indication of this disease, especially its meta-object protocol's fear of closed implementations.

[2]Other Lisp-based object-oriented programming systems include *Flavors* [Cannon82] [Moon86], *CommonLoops* [Bobrow86], *CommonObjects* [Snyder86a], *Oaklisp* [Lang86], *GLISP* [Novak83], *ObjVlisp* [Cointe87], *EU_LISP* [Queinnec88], *HiOBJ-2* [Yasumura90], *TAO* [Takeuchi90], and *EusLisp* [Matsui90].

[3]"[CLOS] does not attempt to solve problems of encapsulation or protection" [DeMichiel89,s.2.2].

ACM OOPS Messenger **2**, 4 (October 1991), 4-15.

ETIOLOGY

CLOS has been criticized on several accounts. First, its size is quite large, which not only increases the size of the Common Lisp reference manual [Steele90] by a substantial amount, but also increases the size and cost of Common Lisp implementations. Second, its extreme generality leads to slow implementations, and the complexity involved in achieving reasonable speed is quite high. Third, its metacircular definition through the use of first-class "classes", "slots", "generic functions" and "methods" seems to guarantee that the previously-mentioned inefficiencies may be impossible to avoid, because many implementation decisions are built into the protocols.

We are interested in object-oriented Lisp systems which produce compiled code which is capable of high performance when executed in a delivery environment [Baker90b].[4] A delivery environment differs from a development environment chiefly by its lack of software development tools such as compilers, linkers and debuggers. Therefore, some techniques for achieving high performance [Deutsch84] [Ungar87] [Chambers89a,b] [Chambers90], are not applicable to our problem because they require the possibility of compiling additional code at run-time.

CLOS is a true object-oriented language, offering objects with "object identity" [Baker93], a multiple-inheritance class hierarchy and polymorphic "generic" (virtual) functions whose methods depend upon the classes of their arguments. Unlike Simula, Beta, C++ [Stroustrup86] and other "compile-time" object-oriented systems, CLOS has a system of first-class classes, which are accessible by a running program, and extreme flexibility, allowing classes to be redefined and moved, and allowing instances to dynamically change their class. In this way CLOS is more like Smalltalk [Goldberg83], Oaklisp [Lang86], and ObjVlisp [Cointe87], but offers additional flexibility in its generic functions, which can choose their methods based on multiple arguments instead of a single argument.

Common Lisp, on which CLOS is based, has already been criticized for its emphasis on generality and flexibility rather than execution speed. The "dynamic" or "weak" typing of Lisp does not allow the compiler to unambiguously determine the types of objects, and therefore its execution speed tends to be somewhat slower than "strongly-typed" languages such as Pascal, Ada and C. Type determination can be enhanced by type *declarations* [Steele90,ch.9] [Hagmann83] [Johnson86] [Suzuki81] and by clever type *inference* [Schwartz75] [Kaplan80] [Suzuki81] [Borning82] [Johnson86] [Ma90] [Baker90d,f], which can provide substantial speedups in compiled code.

The introduction of CLOS polymorphic "generic" functions into Common Lisp greatly magnifies the type determination problem, because any ambiguity regarding the type of an argument can translate into ambiguity regarding which method to use. Conversely, the ambiguity regarding which methods are called propagates and increases the ambiguity for other calls to other generic functions.[5] Whenever there is ambiguity over the type (hence class) of an argument, some portion of the method selection logic must be executed at run-time. If the method selection logic is quite complex, as it is in CLOS, then the execution time penalties can be quite large.

Currently, there are two major approaches used to speed method selection in dynamically-typed object-oriented languages like Smalltalk and CLOS. The first method utilizes a *cache* [Conroy83] which compares the classes of the arguments with the classes of arguments stored in the cache, and uses this information to retrieve the correct method. In the case of a cache miss, then an appropriate method must be selected or constructed. The second method utilizes a form of "partial evaluation" which is really a form of "partial compilation"; methods not in the cache may not even be compiled until such time as that particular method is needed [Chambers89a,b].

---

[4] *GLISP* [Novak83] and *CommonObjects* [Snyder86a] have some of the same goals.
[5] *Within* a method, however, argument specializers can provide valuable type information.

Dynamic methods for achieving generic function application speed can work well in the appropriate environment. Researchers have reported speeds of generic function application which average nearly as fast as a non-generic function application [Kiczales90a]. These techniques are not appropriate, however, for some environments, due to their stochastic behavior; e.g., some real-time applications cannot survive even an occasional method cache miss. These dynamic caches also become a source of bottleneck rather than speed in a multiprocessor environment, because they become a shared database, which—like all shared databases—must be protected by a synchronization protocol.

Our interest in more static mechanisms for method determination is four-fold. First, we would like to achieve compiled speeds that are competitive with more traditional compiled, but not object-oriented, languages. Secondly, we would like to determine methods at compile time so that more traditional type inference can reduce type ambiguity. Thirdly, the compile-time method determination allows for the *inlining* of small methods such as accessor functions (which do not require special treatment in our system); inlining can achieve far more efficiency than the traditional goal of equivalent function application and generic function application execution times. Lastly, we would like to gain a deeper understanding of the occasions and context of code reuse in the form of methods, so that more efficient object-oriented systems can be designed in the future.

Properties of the standard CLOS Class Precedence Ordering

One of the fundamental ideas in CLOS is the notion of a *linearly-ordered* "class precedence list". Most current object-oriented systems which utilize multiple inheritance rely only on the *partial* ordering induced by the direct inheritance links. Totally ordering the class partial order is controversial, as methods do not communicate directly with methods for their direct superclasses [Snyder86b]. We do not address the larger question of whether the total ordering of superclasses is a good programming style [Snyder86b], or whether inheritance itself is good programming style [Lieberman86] [LaLonde86], but rather address the pragmatic issue of which linear ordering might lead to the highest efficiency.

A property of partial orders is that they can be consistently extended into a total order, but this total order is not unique unless the partial order is already a total order. The partial (rather than total) ordering of classes is an opportunity for optimization, because an implementation is then free to choose its own total ordering when it is required, rather than being given a total ordering within which there is no freedom to reorder classes. CLOS, however, closes off this opportunity for optimization.

For the set of classes which are superclasses of a given class, CLOS carefully defines a total ordering consistent with the class partial order called the *class precedence list* for the given class. The computation of this total ordering depends only upon the superclasses of the given class, and not upon the class hierarchy as a whole; thus, once all of the superclasses of a class are known, the class precedence list for the given class is well-defined. Furthermore, the definition of additional classes cannot affect an already computed class precedence list, so long as the class hierarchy above the given class is not modified (e.g., by using `change-class`).

CLOS carefully defines the ordering of the class precedence list due to the existence of `:before` and `:after` methods for generic functions (among other reasons). Since the computed results of `:before` and `:after` methods are ignored, they must achieve all of their results through side-effects. By ordering `:before` methods in class precedence list order, and `:after` methods in reverse class precedence list order, the effects of such `:before` and `:after` methods become well-defined. Since many CLOS implementations compile "combination" methods by concatenating the various `:before` or `:after` methods together, the definition of the class precedence list using only the local properties of the particular class assures the implementation that the additional of any new classes cannot cause any changes to previously-computed class precedence lists, and therefore previously-computed "combination" methods need not be recompiled.

While such a *local* treatment for class precedence lists is laudable for an incremental-development prototyping system,[6] it has very negative consequences for a more compiler-oriented system. To fully appreciate the negative consequences of the local calculation of the class precedence lists, we need to give several examples, which will illustrate several non-obvious properties of the CLOS ordering rules, some of which are presented here for the first time.

Definition. We denote the class precedence list of A by `cpl(A)`.

Theorem 1. If we form the class C from the direct superclasses A and B (in that order), then `cpl(C)` is not necessarily consistent with `cpl(A)`.

Proof. The system generated by the following class definitions is a counterexample, because `y` precedes `x` in `cpl(a)`, but not in `cpl(c)`.

```
(defclass z () ())              ; cpl(z) = (z ...)
(defclass x (z) ())             ; cpl(x) = (x z ...)
(defclass y () ())              ; cpl(y) = (y ...)
(defclass b (y) ())             ; cpl(b) = (b y ...)
(defclass a (b x) ())           ; cpl(a) = (a b y x z ...)
(defclass c (a b x y) ())       ; cpl(c) = (c a b x z y ...)
```

Corollaries. 1) It is impossible in CLOS to form a global class precedence list, of which all local class precedence lists are subsequences. 2) Class precedence lists cannot be computed by a simple merging of the class precedence lists of the direct superclasses. 3) Class precedence list structures cannot generally share tails with the class precedence list structures of their superclasses. 4) The combined `:before` method for a class cannot simply tail-call the combined `:before` method for one of its superclasses. 5) A subclass can *reorder* the applicable methods of its superclasses even when it does not itself define any methods![7] 6) A generic function may require O(n!) method combinations for a class having only O(n) methods defined. 7) If the ordering of an object's slots are determined by the ordering of the class precedence list, then the slot accessor functions become gratuitously polymorphic.[8]

Proof of 4). If there are separate `:before` methods defined for a generic function for every class, then the problem of constructing a `:before` combined *method* is isomorphic to the problem of constructing a class precedence *list*.

Proof Sketch of 6). We can define classes with O(n) unordered superclasses, and provide O(n!) subclasses whose class precedence lists reorder those superclasses. Even though there are no methods defined on the subclasses, we can be forced to make O(n!) different method combinations for the superclasses.

Theorem 2. Class construction is not associative; the classes `abc1` and `abc2` are not isomorphic:

```
(defclass ab (a b) ())
(defclass abc1 (ab c) ())
(defclass bc (b c) ())
(defclass abc2 (a bc) ())
```

[6]For example, local treatment allows class inheritance to be *finalized* as soon as all superclasses have been defined.

[7]I'll bet that there exists a CLOS implementation that fails on this property.

[8]Slot ordering can be decoupled from the class precedence ordering to achieve substantial storage savings [Pugh90].

Proof. Consider the orderings induced by the following definitions of `a`, `b` and `c`. The formation of `abc1` succeeds, but the formation of `abc2` fails due to the violation of a precedence constraint.

```
(defclass a () ())                    ; cpl(a) = (a ...)
(defclass b () ())                    ; cpl(b) = (b ...)
(defclass c (a) ())                   ; cpl(c) = (c a ...)
(defclass ab (a b) ())                ; cpl(ab) = (ab a b ...)
(defclass abc1 (ab c) ())             ; cpl(abc1) = (abc1 ab c a b ...)
(defclass bc (b c) ())                ; cpl(bc) = (bc b c a ...)
(defclass abc2 (a bc) ())             ; fails.
```

Theorem 3. The class constructed by `(defclass a (b c d) ...)` is isomorphic to the class constructed by the sequence `(defclass cd (c d) ())` and `(defclass a (b cd) ...)`.

Proof Sketch. The only difference between the two definitions is the presence of the intermediate class `cd` in the class precedence lists of `a` and all of its subclasses; if `cd` is simply deleted from these lists, then they are identical to the first case. The presence of the intermediate class `cd` does not change the inherited slots, and so long as no methods are defined with `cd` as a specializer, then the presence of `cd` cannot affect either method selection or method combination, and hence the behavior remains the same. Intuitively, the `b-c` precedence constraint is converted into the pair of constraints `b-cd` and `cd-c`.

Corollary—"Double Inheritance in CLOS is Universal". CLOS class construction for multiple inheritance can be emulated by class construction using only single and double inheritance. The right-associative reading of class construction is analogous to Chomsky Normal Form for context-free grammars.

Theorem 4. The classes `(defclass a (b c) ...)` and `(defclass a' (b d c) ...)` are isomorphic when `(b d c)` is a subsequence of the class precedence list of `a`. (In other words, including additional direct superclasses which are mentioned in precedence order cannot change the behavior of a class.)

Proof Sketch. Since the slots of a class are formed as the *union* of the slots contributed by the superclasses, there is no harm in including a superclass twice. The behavior of a class is determined by the methods of the class and its superclasses, as interpreted by the class precedence list. Since we have introduced no new classes or methods, and since the class precedence lists are the same, the situations are isomorphic.

We believe that while the calculation of the class precedence list defined by the CLOS standard was well-intentioned, its specificity is gratuitous, because 1) no well-structured program should depend upon the detailed precedence order over and above those ordering constraints already evident in the construction order and class partial order; and 2) such an order dependence would be a violation of modularity for the program [Snyder86b]. Therefore, we recommend that the CLOS standard back down from its rigid class precedence order,[9] and simply require that the local class precedence orders be consistent with the construction order and the class partial order, and that each local class precedence order be a subsequence of a single global total order.[10] This new rule can dramatically simplify a compiled CLOS implementation, and while

[9]It is interesting that the CLOS precedence ordering is substantially different from that in "old" Flavors, which occasionly violated the class hierarchy partial order [Cannon82]. Such differences lend credence to our contention that the current CLOS precedence ordering is not the last word on the subject of precedence orders. [Stefik86], [Snyder86b] and [Kempf87] discuss rationales for precedence orders.

[10]The `pastry` and `pie` classes in [Steele90,s.28.1.5.2] become incompatible as a result, but that example is already quite contrived.

contrived software can detect the new orderings, well-constructed, modular software will not notice the difference.[11]

TREATMENT: STATIC CLOS

The process of compilation involves the discovery of facts about a program which are independent of the data that will be fed to the program. The greater the number of these "universal" facts, the higher the level of optimization that is possible. Object-oriented languages, however, are the antithesis of optimizing compilers. This is because object-oriented languages (especially those with "meta-object protocols" [Kiczales90b]) attempt to defer decisions as long as possible, usually until run-time. While such "late binding" characteristics are admirable for software prototyping systems, they are not appropriate for "delivery" systems, which must be debugged, optimized, and then "buttoned up".

Since we are most interested in high-performance delivery capabilities, we have developed a subset of CLOS which we call *Static CLOS*, which fixes many decisions at compile-time. The earlier binding of Static CLOS allows for a much greater level of compile-time optimization than would be possible under full CLOS. We have carefully chosen the characteristics to be fixed at compile time, however, so that most traditional object-oriented programming styles are preserved. In making the choices for Static CLOS, we have appealed to the experiences of other compiled object-oriented languages, such as Simula, Ada and C++, so that those users would not find our choices confining.

1. An Object's Class is fixed at Instantiation Time

Static CLOS first requires the class of an object to be constant throughout its lifetime. Those from a traditional compiled-language background will find this choice non-controversial, but those from a database background might find this unacceptable. We make this requirement because it is usually true, and it makes the job of performing type inferencing much, much easier.

An alternative would be to separate objects into those whose types are fixed at compile time and those whose types can be varied at compile time and to focus most of the optimization on the first sort. Not only is this scheme more complex, but it conflicts with our whole philosophy of types, which we assert are statically-defined subsets of run-time objects.

Some would argue that changing the class of an object at run-time is necessary for the proper modeling of the real world; e.g., some might model state changes in real-world objects as changes in the class of the object. Philosophers have argued that objects have certain innate, fixed attributes or characteristics that are inseparable from their object identity, while other attributes and characteristics can change dynamically. Philosophers have called these necessary attributes of objects "formal properties", while those attributes which can be changed are called "material properties" [Hughes68,p.184]. Any material properties should be grouped into the object's states, leaving the object's identity and class fixed. Thus, the use of the class system which organizes representations and code-sharing to model real-world state changes is a nasty pun.

Others would argue that changing the class of an object is necessary for "database reorganization", in which a later "database schema" would use a more complex model requiring additional fields or "slots" for an object [Lerner90] [Penney87] [Skarra86]. While this sort of change is on more solid theoretical ground, one must still protest that object-oriented classes organize representations and code-sharing, and a change of representations should involve a

---

[11]The CLOS "metaobject protocol" [Kiczales90b] [des Rivières90] may have to be modified to allow such global precedence calculations. It currently attempts to compute a precedence list as soon as all superclasses are known. The global class precedence list, of which all class precedence lists will be a subsequence, will not be known, however, until all classes have been defined. An alternative would be to guess the global precedence list, and be prepared to fix things if the guess proves wrong.

change of class and a change of object identity. We feel that database reorganization is better served by the notion of an "abstract data type", which allows for a diversity of *representations* within a single user-visible *specification* type. We discuss the issue of fixed object type at greater length in [Baker90c,Baker93].

2. The Class Hierarchy is fixed at compile time

Static CLOS next requires that the class hierarchy be fixed at compile time.[12] This means that the representations of the class instances can also be fixed at compile time, just as they they could with structures and classes in other compiled languages such as Simula and C++. Statically fixing the class hierarchy offers the possibility of performing class inference as well as more traditional type inference [Schwartz75] [Kaplan80] [Suzuki81] [Borning82] [Johnson86] [Ma90] [Baker90c,d,f], so that some information is available to restrict the ambiguity as to which methods are applicable at particular generic function calls.

Statically fixing the class hierarchy is controversial, because it eliminates the possibility of dynamic "schema changes", which reorganize the structure of the knowledge embedded in the class hierarchy [Skarra86]. While such reorganizations are common during the development of an application's prototype, they are unlikely to be required after delivery. The use of a persistent object-oriented database may require such a reorganization, but even in such a database, the rearrangement of the class hieararchy is a traumatic and rare occurrence. If recompilation is required in such a situation, it would not be onerous.

The fixing of the class hierarchy at compile time also allows for a significant simplification in the creation of the class hierarchy—one could conceivably require classes to be defined in topological order. Since the class hierarchy must be a partial order [Steele,s.28.1.2], and since every partial order can be extended into a consistent total order, the requirement for no "forward references" in the class hierarchy cannot reduce the functionality of the system.[13] In fact, a consistent ordering of the class hierarchy in the source file from more general to more specific could be a significant documentation aid.

We argued above that all class precedence lists should be subsequences of a global class precedence list which is consistent with both the hierarchy precedence order as well as the construction ordering constraints.[14] With this additional assumption, we can equate the class definition ordering with the global class precedence ordering, and the programmer thus indicates by the ordering of class definitions in the source code exactly what the global precedence ordering should be.[15] "Mixin" classes [Cannon82] [Stefik86] [Bracha90], which typically modify the behavior of "base" classes, would appear later in the source code (and earlier in the class construction lists), thus allowing them to replace base class behaviors.[16]

3. Generic Functions, including Slot Accessor Functions

CLOS generic functions cannot be fixed at compile time due to the existence of *anonymous* generic functions. Static CLOS does, however, require that the classes required by generic

---

[12]While the class hierarchy must be fixed at compile time, there is no requirement that the class definitions be explicitly written out in a source file. They could be generated by a macro, for example.

[13]While there need be no forward references in the class precedence list, there may still be forward references in the slot definitions, because a slot can hold a reference to an arbitrary object, including one whose class has not yet been defined.

[14]This global ordering can be used to order the bits in the type inferencing bit-vectors [Baker90b,c].

[15]Some would say that a global linear precedence ordering violates the philosophy of object "independence", but this violation is already entailed by the presence of unrestricted side-effects in `:before` and `:after` methods. Our global precedence ordering is simply a way to make these side-effects predictable.

[16]Some advocated uses of the `:before`, `:after` and `:around` methods of CLOS include the management of multiprogramming *locks* [Keene89]. Locking protocols are required in order to avoid deadlock. One such protocol involves an arbitrary, but constant, ordering of lock acquisition; our *global* precedence order will guarantee that this protocol ordering is never violated.

functions are defined before the generic function itself is defined. Generic functions implement polymorphic behavior in CLOS by "dispatching" to different methods based on the classes of the arguments. The goal of Static CLOS is to reduce CLOS generic functions to traditional non-generic functions which include an explicit type dispatch; type inference can then be used to simplify (and often remove, especially in the case of slot accessor functions) the type dispatch [Baker90d]. Consider the following CLOS generic function:

```
(defgeneric foo (x y))
(defmethod foo ((x integer) y) (foo-integer x y))
(defmethod foo ((x character) y) (foo-character x y))
(defmethod foo ((x (eql 3)) y) (foo-3 x y))
(defmethod foo :before (x y) (format t "~%Entering foo..."))
(defmethod foo :after ((x character) y) (format t "~%Did foo-char"))
```

We wish to produce a non-CLOS implementation similar to the following:

```
(defun foo (x y)
 (format t "~%Entering foo...")
 (case x                                          ; first "eql" dispatch
  (3 (foo-3 x y))
  (t (typecase x                                  ; then "type" dispatch
      (integer (foo-integer x y))
      (character (multiple-value-prog1 (foo-character x y)
                                       (format t "~%Did foo-char")))
      (t (no-applicable-method #'foo x y))))))
```

Intuitively, we replace the implicit method dispatch code with explicit dispatch code, which can be optimized by traditional optimization techniques, including the use of a compiler-generated hash table for performing the dispatch. There have been previous object-oriented extensions of Lisp which achieved nearly this paradigm of simplicity and efficiency [Novak83]; the complexity and dynamism of CLOS, however, makes our task more difficult.

While the idea is simple, the translation is complicated by the presence of CLOS method combinations. Unlike traditional Smalltalk, which runs the most specific method, and perhaps calls upon the method of its superclass, CLOS allows for `:before`, `:after` and `:around` methods, in addition to "vanilla" *primary* methods.[17] We need to translate generic functions in such a way that the correct behavior is produced, regardless of the details of the method code.

Methods are not themselves complete functions, because they assume a binding for two function names—`call-next-method` and `next-method-p`, which are to be bound as if by `flet`. These functions are used by the method programmer to check, and possibly call, the next applicable method on the precedence list. Generic functions must somehow "glue" these methods together into a traditional Lisp function. For primary, `:before` and `:around` methods, the ordering is identical to the precedence ordering, while for `:after` methods, the ordering is the reverse. Since reverse-ordered precedence lists cause an unintended dependence of a class on its subclasses, we provide a structure wherein `:before` and `:after` methods are called in a manner similar to `:around` methods. This scheme utilizes the run-time stack to remember to call the `:after` methods in the correct order.[18] Thus, we compile the `:before` and `:after` methods into a single closure, the primary method into another closure, and the `:around` method into third closure.[19]

---

[17]We do not discuss here method combinations like + and `nconc`, which combine the results of all like methods for all of superclasses; these combinations are simplified (but also modified if not commutative and associative) by our global class precedence list.

[18]Smalltalk programmers use an equivalent scheme to emulate `:before/:after` methods.

[19]If one of these kinds of methods is not present, then a suitable "null method" is used; it will be optimized away later by the compiler.

The code for the generic function itself is a simple type dispatch on the type(s) of the required argument(s)[20]; the arms of this dispatch are calls to class-specific functions. We must include in the type dispatch every class that is mentioned as a specializer on one of the methods for the generic function.[21] Each class-specific function glues together the applicable method closures in precedence order using lambda-binding techniques; straight-forward beta reduction ("inlining") optimizations on these glue functions substantially reduce the function-calling overhead. In particularly simple (but common) cases, the class-specific function could have simply called the class-specific function for a superclass in traditional Smalltalk "send-super" fashion; an optimization to do this would save space by sharing compiled code (not just source code).

The effectiveness of Static CLOS depends critically upon the ability of type inference to reduce the number of method choices—hopefully to a single method. Our type inference techniques [Baker90c,d,f] are not based on ML-style unificational type inference [Milner78] [Suzuki81] [Cardelli89], but on dataflow techniques [Kaplan80] similar to, but more powerful than, traditional *abstract interpretation* [Cousot77]. As a result, our type inferencer handles *full* Common Lisp-84, with all of its *ad hoc* polymorphism, and includes the ability to predict a more precise type within the arm of a conditional or dispatch than in the expression as a whole. This ability to utilize the information generated by a predicate or a dispatch is well-suited to the job of disambiguating method selection at compile time. The generality of the Common Lisp type system and this type inferencer means that Static CLOS is remarkably free from the well-known typing restrictions of other compile-time object-oriented languages [Cardelli89] [Madsen90]—e.g., Simula, Beta, C++, etc. The significant restrictions of Static CLOS are those already mentioned—an object has a constant type and class, the class system is completely known at compile time, and local class precedence lists are consistent with a global linear precedence list. In particular, Static CLOS makes no restriction on first-class generic functions, and includes the ability to create anonymous generic functions at run-time, as well as the ability to locally add a method within a static context.[22]

CONCLUSIONS AND FUTURE WORK

We have shown how the Common Lisp Object System (CLOS) can be implemented by techniques that can allow a number of methods to be determined at compile time through the use of a sophisticated type inference facility [Baker90c,d,f], and therefore approach the efficiency of "strongly-typed" object-oriented programming systems such as Beta, Eiffel, and C++. We are attempting to achieve many of the optimizations of other CLOS implementations (e.g., [Cyphers90]) automatically, through general techniques such as constant propagation and inlining, rather than through complex *ad hoc* optimizations. Due to the extreme generality of full CLOS, we have focussed on a subset which we call *Static CLOS*, in which opportunities for these optimizations are much higher, and which achieve in many cases compile-time method selection. In addition to requiring that objects have constant type and class, and requiring that the class hierarchy be known at compile time, Static CLOS also makes a slight modification in CLOS's scheme for computing class precedence lists. We have argued that our modified

---

[20]If there are `eql` specializers, we must dispatch on them first.

[21]Static CLOS provides for *uninstantiable* classes [McAllester86], which can substantially reduce the sizes of these dispatches. We note that to implement *full* CLOS, with its *local* class precedence lists, then the generic dispatch would, in general, have to include *every* defined subclass that could have instances, not just those which have methods. A Corollary to Theorem 1 shows that this can exponentially blow up the number of cases that need to be considered. This unfortunate state of affairs results from the ability of a subclass to not just call the methods of its superclasses, but also to *reorder* them.

[22]Such late-bound code is likely to be less precisely typed, however, and may therefore be less efficiently compiled.

ordering drastically reduces the complexity of the implementation and improves the understandability of the resulting system.[23]

We are preparing some benchmark tests to compare the speed of our Static CLOS with the generally-available "PCL" implementation of CLOS. We have scanned the CLOS literature (e.g., [Keene89]), looking for examples in which our modified class precedence lists would fail, and have as yet found only the non-commutative and/or non-associative method combinations `append`, `list`, `nconc`, `progn`, whose users are already on notice regarding ordering dependencies. We have utilized a similar global linear class ordering for a prototype of the DARPA Pilot's Associate object library, and have yet to find the global linear ordering confining. Additional examples must be analyzed.

We have developed a Common Lisp-to-Ada translator to demonstrate the power of Lisp as a prototyping/specification language for Ada [Baker91a]. We have proposed the use of Static CLOS in conjunction with this translator to demonstrate the ability to translate an existing large CLOS system into Ada.

In implementing Static CLOS, we have bypassed much of the CLOS *metaobject protocol* (or "MOP" [Kiczales90b]), which offers the ability to customize CLOS for particular applications. Whether we could have used the CLOS MOP to implement Static CLOS requires more research; our current feeling is that MOP is too oriented towards a run-time implementation.

REFERENCES

Adams, N., and Rees, J. "Object-Oriented Programming in Scheme". *Proc. 1988 ACM Conf. on Lisp and Funct. Progr.*, Snowbird, UT, July 1988,277-288.

Atkinson, R.G. "Hurricane: An Optimizing Compiler for Smalltalk". *Proc. OOPSLA'86, Sigplan Not. 21*,11 (Nov. 1986),151-158.

Baker, Henry. "Unify and Conquer (Garbage, Updating, Aliasing, ...) in Functional Languages". *ACM Lisp and Funct. Progr. Conf.*, June, 1990,218-226.

Baker, Henry. "The NIMBLE Project—An Applications Compiler for Real-Time Common Lisp". *Proc. InfoJapan'90*, Info. Proc. Soc. Japan, Tokyo, Oct. 1990

Baker, Henry. "A Decision Procedure for Common Lisp's `SUBTYPEP` Predicate". *Lisp and Symbolic Computation*, to appear.

Baker, Henry. "The Nimble Type Inferencer for Common Lisp-84". Submitted to *ACM TOPLAS*, 1990.

Baker, Henry. "Equal Rights for Functional Objects". ACM *OOPS Messenger 4*,4 (Oct. 1993), 2-27.

Baker, Henry. "Integrated Static Inference". Nimble Technical Report, 1990.

Baker, Henry. "Structured Programming with Limited Private Types in Ada: Nesting is for the Soaring Eagles". *ACM Ada Letters XI*,5 (July/Aug. 1991),79-90.

Baker, Henry. "Object-Oriented Programming in Ada83, or, Rehabilitating Genericity". *ACM Ada Letters XI*,9 (Nov./Dec. 1991),116-127.

Ballard, S., and Shirron, S. "The Design and Implementation of VAX/Smalltalk-80". [Krasner83],127-150.

Ballard, M.B., Maier, D, and Wirfs-Brock, A. "QUICKTALK: A Smalltalk-80 Dialect for Defining Primitive Methods". *Proc. OOPSLA'86, Sigplan Not. 21*,11 (Nov. 1986),140-150.

Bobrow, D., *et al.* "CommonLoops: Merging Lisp and Object-Oriented Programming". *Proc. OOPSLA'86, Sigplan Not. 21*,11 (Nov. 1986),17-29.

Bobrow, D.G., and Kiczales, G. "The Common Lisp Object System Metaobject Kernel: A Status Report". *Proc. 1988 ACM Conf. on Lisp and Funct. Progr.*, Snowbird, UT, July 1988,309-315.

Borning, A., and Ingalls, D. "A Type Declaration and Inference System for Smalltalk. *ACM POPL 9* (1982),133-141.

Bracha, Gilad, and Cook, William. "Mixin-based Inheritance". *Proc. OOPSLA'90, Sigplan Not. 25*,10 (Oct. 1990),303-311.

Bush, W.R., *et al.* "Compiling Smalltalk-80 to a RISC". *Proc. ASPLOS II, Sigplan Not. 22*,10 (Oct. 1987),112-116.

[23]The modified class precedence rule is not an essential part of Static CLOS, in that Static CLOS still generates correct code for the standard rule. Such code, however, may be exponentially larger than the code produced using the modified rule.

Cannon, Howard. "Flavors: A non-hierarchical approach to object-oriented programming". Unpublished manuscript, 1982.

Cardelli, Luca. "Typeful Programming". Tech. Report No. 45, DEC Systems Research Ctr., May, 1989,63p.

Caudill, Patrick, and Wirfs-Brock, Allen. "A Third Generation Smalltalk-80 Implementation". OOPSLA'86, also Sigplan Not. 21,11 (Nov. 1986),119-129.

Chambers, C., and Ungar, D. "Customization: Optimizing Compiler Technology for SELF, A Dynamically-Typed Object-Oriented Programming Language". *Proc. PLDI'89, Sigplan Not. 24*,7 (July 1989),146-160.

Chambers, C., Ungar, D., and Lee, E. "An Efficient Implementation of SELF, a Dynamically-Typed Object-Oriented Language Based on Prototypes". *Proc. OOPSLA'89, Sigplan Not. 24*,10 (Oct. 1989),49-70.

Chambers, C., and Ungar, D. "Iterative Type Analysis and Extended Message Splitting: Optimizing Dynamically-Typed Object-Oriented Programs". *Proc. PLDI'90, Sigplan Not. 25*,6 (June 1990),150-164.

Cointe, Pierre. "Metaclasses are First Class: The ObjVlisp Model". *Proc. OOPSLA'87, Sigplan Not. 22*,12 (Dec. 1987),156-167.

Connor, R.C.H., et al. "An Object Addressing Mechanism for Statically Typed Languages with Multiple Inheritance". *Proc. OOPSLA'89, Sigplan Not. 24*,10 (Oct. 1989),279-285.

Conroy, T.J., and Pelegri-Llopart, E. "An Assessment of Method-Lookup Caches for Smalltalk-80 Implementations". [Krasner83],239-247.

Cousot, P., and Cousot, R. "Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints". *ACM POPL 4* (1977),238-252.

Cyphers, D. Scott, and Moon, David. "Optimizations in the Symbolics CLOS Implementation". *Proc. 3rd CLOS Users and Implementors Workshop, in conj. w/OOPSLA'90* (Oct. 1990),18-23.

DeMichiel, L.G. "Overview: The Common Lisp Object System". *Lisp and Symbolic Comput. 1*,3/4 (Jan. 1989),227-244.

des Rivières, Jim, and Kiczales, Gregor. "The Art of the Metaobject Protocol: A backstage look at CLOS implementations". Unpublished manuscript, Xerox Palo Alto Research Center, Oct. 15, 1990,203p.

Deutsch, L.P., and Schiffman, A.M. "Efficient Implementation of the Smalltalk-80 System". *Proc. 11'th ACM POPL*, Salt Lake City, UT, Jan. 1984,297-302.

Dixon, R., *et al.* "A Fast Method Dispatcher for Compiled Languages with Multiple Inheritance". *Proc. OOPSLA'89, Sigplan Not. 24*,10 (Oct. 1989),211-214.

Dussud, Patrick. "TICLOS: An Implementation of CLOS for the Explorer Family". *Proc. OOPSLA'89, Sigplan Not. 24*,10 (Oct. 1989),215-219.

Goldberg, A., and Robson, D. *Smalltalk-80: The Language and its Implementation*. Addison-Wesley, 1983.

Hagmann, Robert. "Preferred Classes: A Proposal for Faster Smalltalk-80 Execution". [Krasner83],323-330.

Hughes, G.E., and Cresswell, M.J. *An Introduction to Modal Logic*. Methuen and Co., London, 1968.

Johnson, Ralph E. "Type-Checking Smalltalk". *Proc OOPSLA'86, Sigplan Not. 21*,11 (Nov. 1986),315-321.

Kaplan, Marc A., and Ullman, Jeffrey D. "A Scheme for the Automatic Inference of Variable Types". *J. ACM 27*,1 (Jan. 1980),128-145.

Keene, S.E. *Object-Oriented Programming in Common Lisp*. Addison-Wesley, Reading, MA, 1989.

Kempf, James, *et al.* "Experience with CommonLoops". *Proc. OOPSLA'87, Sigplan Not. 22*,12 (Dec. 1987),214-226.

Kiczales, G. and Rodriguez, L. "Efficient Method Dispatch in PCL". *Proc. 1990 ACM Conf. on Lisp and Funct. Progr.*, Nice, France, June 1990,99-105.

Kiczales, G., and Bobrow, D.G. "Common Lisp Object System Specification 3: Metaobject Protocol". Unpublished manuscript, Xerox Palo Alto Research Center, July 30, 1990,126p.

Kim, Won, and Lochovsky, F.H., *eds*. *Object-Oriented Concepts, Databases and Applications*. Addison-Wesley, Reading, MA, 1989.

Krasner, Glenn, *ed*. *Smalltalk-80: Bits of History, Words of Advice*. Addison-Wesley, Reading, MA, 1983.

LaLonde, Wilf, Thomas, Dave A., and Pugh, John R. "An Exemplar-Based Smalltalk". *OOPSLA'86, also Sigplan Not. 21,*11 (Nov. 1986),322-330.

Lang, K.J., and Pearlmutter, B.A. "Oaklisp: an Object-Oriented Scheme with First Class Types". *Proc OOPSLA'86, Sigplan Not. 21*,11 (Nov. 1986),30-37.

Larus, James Richard. *Restructuring Symbolic Programs for Concurrent Execution on Multiprocessors*. Ph.D. Thesis, UC Berkeley, also Report No. UCB/CSD/89/502, May, 1989.

Lerner, Barbara Staudt, and Habermann, A. Nico. "Beyond Schema Evolution to Database Reorganization". *OOPSA/ECOOP'90, also Sigplan Not. 25*,10 (Oct. 1990),67-76.

Lieberman, Henry. "Using Prototypical Objects to Implement Shared Behavior in Object-Oriented Systems". *OOPSLA'86, also Sigplan Not. 21*,11 (Nov. 1986),214-223.

Ma, Kwan-liu, and Kessler, Robert R. "TICL—A Type Inference System for Common Lisp". *Software Proc. & Exper. 20*,6 (June 1990),593-623.
Madsen, Ole Lehrmann, et al. "Strong Typing of Object-Oriented Languages Revisited". *ACM OOPSLA/ECOOP'90, also Sigplan Not. 25*,10 (Oct. 1990),140-150.
Matsui, T., and Inaba, M. "EusLisp: An Object-Based Implementation of Lisp". *J. Info. Proc. 13*,3 (1990),327-338.
McAllester, David, and Zabih, Ramin. "Boolean Classes". *Proc. OOPSLA'86, Sigplan Not. 21*,11 (Nov. 1986),417-423.
Mellender, F, Riegel, S., and Straw, A. "Optimizing Smalltalk Message Performance". [Kim89],423-450.
Milner, Robin. "A Theory of Type Polymorphism in Programming". *JCSS 17* (1978),348-375.
Miranda, E. "BrouHaHa—A Portable Smalltalk Interpreter". *Proc. OOPSLA'87, Sigplan Not. 22*,12 (Dec. 1987),354-365.
Moon, David A. "Object-Oriented Programming with Flavors". *Proc. OOPSLA'86, Sigplan Not. 21*,11 (Nov. 1986),1-8.
Moon, David A. "The Common Lisp Object-Oriented Programming Language Standard". [Kim89],49-78.
Novak, Jr., Gordon S. "Data Abstraction in GLISP". *Proc. SIGPLAN'83, Sigplan Not. 18*,6 (June 1983),170-177.
Penney, D. Jason, and Stein, Jacob. "Class Modification in the GemStone Object-Oriented DBMS". *Proc. OOPSLA'87, Sigplan Not. 22*,12 (Dec. 1987),111-117.
Pugh, W, and Weddell, G. "Two-Directional Record Layout for Multiple Inheritance". *Proc. PLDI'90, Sigplan Not. 25*,6 (June 1990),85-91.
Queinnec, C., and Cointe, P. "An Open-ended Data Representation for EU_LISP". *ACM Lisp and Funct. Progr. Conf.* (July 1988),298-308.
Reddy, Uday S. "Objects as Closures: Abstract Semantics of Object-Oriented Languages". *Proc. 1988 ACM Conf. on Lisp and Funct. Progr.*, Snowbird, UT, July 1988,289-297.
Rose, J.R. "Fast Dispatch Mechanisms for Stock Hardware". *Proc. OOPSLA'88, Sigplan Not.23,*11 (Nov. 1988)27-35.
Samples, A.D., Ungar, D., and Hilfinger, P. "SOAR: Smalltalk Without Bytecodes". *Proc. OOPSLA'86, Sigplan Not. 21*,11 (Nov. 1986),107-118.
Schwartz, J.T. "Optimization of very high level languages—I. Value transmission and its corollaries". *J. Computer Lang. 1* (1975),161-194.
Skarra, Andrea H., and Zdonik, Stanley B. "The Management of Changing Types in an Object-Oriented Database". *Proc. OOPSLA'86, Sigplan Not. 21*,11 (Nov. 1986),483-495.
Snyder, Alan. "CommonObjects: An Overview". *Sigplan Not. 21*,10 (Oct. 1986),19-28.
Snyder, Alan. "Encapsulation and Inheritance in Object-Oriented Programming Languages". *Proc. OOPSLA'86, Sigplan Not. 21*,11 (Nov. 1986),38-45.
Steele, Guy L., Jr. *Common Lisp: the Language; 2nd Edition*. Digital Press, Bedford, MA, 1990.
Stefik, Mark, and Bobrow, Daniel G. "Object-Oriented Programming: Themes and Variations". *AI Magazine 6*,4 (Winter 1986),40-62.
Stroustrup, Bjarne. *The C++ Programming Language*. Addison-Wesley, Reading, MA, 1986.
Stroustrup, Bjarne. "Multiple Inheritance for C++". *Proc. EUUG Spring Conf.*, May 1987.
Suzuki, N. "Inferring Types in Smalltalk". *ACM POPL 8* (1981),187-199.
Suzuki, N., and Terada, M. "Creating Efficient Systems for Object-Oriented Languages". *Proc. ACM POPL 11*, Salt Lake City, UT, Jan. 1984,290-296.
Takeuchi, Ikuo. "TAO—A Lisp Dialect with Multiple Paradigms". *J. Info. Proc. 13*,3 (1990),318-326.
Ungar, D. *The Design and Evaluation of a High Performance Smalltalk System*. MIT Press, Camb., MA, 1987.
Ungar, D., and Smith, R.B. "Self: The Power of Simplicity". *Proc. OOPSLA'87, Sigplan Not. 22*,12 (Dec. 1987),227-242.
Yasumura, Michiaki, *et al*. "HiLISP—a Common Lisp with an Optimizing Compiler on a Mainframe". *J. Info. Proc. 13*,3 (1990),296-303.

